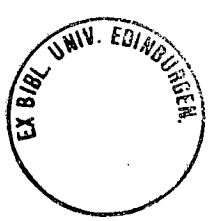
Engineering the Performance of Parallel Applications

Neil Blair MacDonald

Doctor of Philosophy

University of Edinburgh

1996



Abstract

Parallel computing platforms are widely used to run scientific applications. The vast majority of these applications are programmed in an explicitly parallel style. Often the performance of a parallel application is considered only after implementation — in the guise of performance debugging and tuning. Performance engineering approaches incorporate performance analysis into the design phase, using performance data to inform design decisions. This thesis is concerned with performance engineering of parallel applications.

Performance engineering requires accurate predictive models of application performance. The accuracy of micro-analysis techniques for predicting the execution time of sequential code is investigated on a number of representative uniprocessor platforms. This approach is extended to SPMD (Single Program Multiple Data) parallel programs written in a message-passing style using collective communication operations and is used to predict the execution time of commonly occurring parallel application structures. The accuracy of the approach is assessed on a number of representative parallel platforms.

Reasoning about the performance of parallel applications in the absence of contention is straightforward; situations in which all communication serialises can be analysed with a little more sophistication. Reasoning about the effects of contention between these two extreme cases is difficult. Furthermore, allowing point-to-point message-passing operations destroys the assumption of synchrony used to analyse SPMD programs using collective communications. The complexities introduced by these issues inhibit informal reasoning about performance properties of parallel systems. A formal frame-

work for reasoning about the performance of parallel systems is developed, based on a timed process algebra — Eager Timed CCS. Methods for automatically analysing the performance of Eager Timed CCS models are developed and extended to handle abstract Eager Timed CCS models in which time can be represented symbolically. The techniques allow the derivation of parametric expressions for the execution time of models.

Eager Timed CCS descriptions of various parallel applications and parallel platforms are presented, and performance models extracted from them. The applicability of the approach is demonstrated by a number of examples of its use in designing systems.

Acknowledgements

Many people have helped me to write this thesis.

Stuart Anderson supervised my work, and patiently provided sound advice which, had I paid more attention, would have led to a stronger, shorter, and more quickly completed thesis. I am glad that his patience has finally outlasted my procrastination.

I am grateful to Murray Cole, Greg Wilson, Mark Parsons, Sarah Rennie and Andrew Herron for taking the time to read earlier drafts, and for their valuable comments and corrections.

My family, my friends, and my colleagues in Edinburgh Parallel Computing Centre have sustained me on the seemingly perpetual journey towards completing this thesis, and I greatly appreciate their support and their indulgence.

Table of Contents

Chapter 1

Introduction

High performance computing underpins many fields of scientific research and engineering design. In 1995, high performance computing platforms are predominantly, and increasingly, multiple-processor systems. Applications are written in an explicitly parallel style to exploit these parallel systems. Although emerging standards provide software portability across the performance spectrum, the programmer is responsible for selecting one of many possible designs for a parallel application. This thesis develops and evaluates performance prediction techniques which can support the programmer during the design of parallel applications.

We begin by reviewing the trend toward parallelism in high performance computing systems, and the tools used to program parallel computers. We then present performance metrics for parallel applications and review performance evaluation methodologies which have been proposed. This chapter closes with a synopsis of the thesis.

1.1 The Trend Towards Parallelism

Of the 500 most powerful computer systems installed worldwide, as reported in the TOP500 [Dongarra *et al.*, 1995], only 5% are single-processor systems — down from 19% in 1993 [Dongarra *et al.*, 1993]. The ranking of the most powerful single-processor

system has fallen from 35th to 114th in the same period, while the average performance of the TOP500 systems has more than trebled.

Flynn's taxonomy [Flynn, 1972] classifies computer architectures according to their number of instruction and data streams. In Single Instruction Multiple Data (SIMD) architectures, all processors simultaneously execute a single instruction on multiple data whereas Multiple Instruction Multiple Data (MIMD) systems permit each processor to execute its own instruction stream. Both SIMD and MIMD systems appear in the current TOP500 [Dongarra *et al.*, 1995], but the proportion of SIMD systems has fallen to less than 2%.

Turning our attention to MIMD systems, two striking trends can be seen. Firstly, the level of parallelism is increasing. The average number of processors in each system in the TOP500 has doubled from 37 to 74 in the last two years. Secondly, commodity RISC microprocessors are having a massive effect on the high performance computing market. In 1993, only 15% of the TOP500 were based on commodity microprocessors. Today, over 56% use off-the-shelf processors.

To explore these trends further, we classify MIMD systems as multiprocessors (shared memory MIMD systems) and multicomputers (distributed memory MIMD systems). The 251 multiprocessors in 1993's TOP500 [Dongarra *et al.*, 1993] were exclusively vector-parallel systems. While a similar number of multiprocessors appear in the current list, some 40% are based on commodity RISC microprocessors. The vast majority (95%) of the multicomputers in the current TOP500 are also based on commodity microprocessors, although most rely on custom circuitry for inter-processor communication. Increasingly, hardware support for shared memory operations is being provided, although these features are not typically transparent.

The same commodity microprocessors are deployed in parallel systems with more modest performance. All the major workstation vendors now offer multiprocessor systems, and a number of vendors are developing multiprocessor systems based on PC technology. Collections of networked workstations are widely used as virtual multicomputers. High-performance commodity interconnects are being developed which will

allow higher-performance multicomputers to be constructed from off-the-shelf components.

1.2 Programming Parallel Systems

Many multiprocessor systems allow existing sequential applications to be executed on a single processor without modification. These platforms can be used as throughput engines, executing independent applications. Each application takes the same time to execute as it would on a single-processor system, but parallelism enables the system to execute a larger number of applications in a given time. Software packages such as Codine and Condor enable workstation networks to be used in a similar manner [MacDonald and Trew, 1994].

If a single application is to benefit from the higher performance offered by a parallel system, it must be implemented in an appropriate form. To produce a parallel application, the parallelism available in the computation must be identified, the computation must be partitioned or agglomerated into processes, and these processes must be mapped onto processors. In practice, partitioning and mapping are closely intertwined, although mapping is becoming less of an issue for the programmer as higher performance interprocess communications better sustain the illusion of uniform communication latencies.

A large research effort has sought to develop parallelising compilers which can perform these tasks automatically for existing sequential applications [Zima and Chapman, 1990]. However, parallelising compilers for established programming languages have not become effective, principally due to limitations on the amount of parallelism which can be identified in codes. Furthermore, some applications cannot be effectively parallelised unless core algorithms are replaced with alternatives which are more amenable to parallelisation. This task requires careful intervention by a programmer, and is beyond the scope of compiler technology. As a result of parallelising compilers' limitations, new approaches have had to be developed for programming parallel systems.

Parallel programming models and languages can be developed through evolution, by extending established programming models and languages or through revolution, by devising new languages and models. Although many revolutionary parallel programming models and languages have been proposed [Bal *et al.*, 1989], their uptake has been obstructed by the large base of existing code written in traditional imperative languages, such as C and Fortran. The prohibitive cost of re-implementing this code in a new programming language has created inertia in parallel programming language development with the result that multiprocessors and multicomputers are typically programmed in parallel dialects of traditional imperative languages. Attempts by vendors to impose novel programming models on the end user community have been rebuffed. For example, commercial pressures forced Thinking Machines Corporation to provide C and Fortran dialects in addition to their chosen *Lisp, while vendors of transputer-based systems were obliged to provide C and Fortran compilers in addition to occam.

Early parallel computing platforms supported vendor-specific programming dialects which were closely tied to the system's architecture. This presented a barrier to portability which threatened the viability of the industry: established software vendors were unwilling to develop parallel versions of their applications which would only run on one vendor's systems, while vendors needed these applications on their systems to attract a wider user base and a larger market.

When researchers began to address this problem in the mid-1980s, they concentrated on MIMD systems, which were more flexible and more widely used than SIMD architectures. The distributed-memory programming model was chosen because it could be supported more efficiently on multiprocessors than the shared-memory abstraction could be implemented on multicomputers. A number of portable message-passing interfaces were implemented on a range of multicomputer systems in the late 1980s and early 1990s, including P4 [Boyle *et al.*, 1987], PARMACS [Bomans *et al.*, 1990; Hempel *et al.*, 1992], PICL [Geist *et al.*, 1990], TCGMSG [Harrison, 1991], and CHIMP [Bruce *et al.*, 1995; MacDonald, 1995].

Meanwhile, an increasing recognition that workstation networks could be programmed

as multicomputers led to the development of a number of programming environments, of which PVM [Beguelin *et al.*, 1991; Geist *et al.*, 1994] is the most significant. PVM provided a means of writing an explicitly parallel message-passing program for a heterogeneous cluster of workstations. Made freely available, PVM quickly established a large user community world-wide. The importance of workstation clusters as a development route for users was not lost on multicomputer vendors, who began to make their message-passing interfaces available on clusters, and to support PVM implementations on their multicomputers.

By the late 1980s, a number of research groups perceived message-passing to be too low-level a programming model, and sought to establish the data-parallel programming model used on SIMD machines as a portable interface for parallel programming. These groups developed compilers for data parallel languages, such as Fortran D [Hiranandani *et al.*, 1991], Vienna Fortran [Benker *et al.*, 1992] and Dataparallel C [Hatcher and Quinn, 1991], which could effectively target MIMD systems. In 1992, a number of these groups came together with vendors and end users to define High Performance Fortran (HPF), an extension of Fortran 90 which supports portable data parallel programming on a range of architectures. In 1995, robust HPF compilers are beginning to become available for multicomputer systems, including workstation networks. A significant obstacle to the widespread use of HPF is its widely perceived unsuitability for unstructured and irregular problems. The HPF Forum has begun a second round of meetings to try to address these deficiencies, but the prognosis is unclear. Furthermore, it is arguable that HPF is impeded by its basis on Fortran 90, which has not yet been widely adopted by industry as a replacement for Fortran 77. No standard has yet emerged for data parallel programming in C.

It seems likely that message-passing will remain the parallel programming model of choice for many applications for some time.

Inspired by the success of the HPF Forum, a community of researchers, vendors and end users came together in 1993 to form the Message Passing Interface Forum, with a view to defining a standard message passing interface (MPI). A draft definition was issued

the following year, and in 1995 public-domain or vendor-supplied implementations of MPI are available on almost all high performance computing platforms from workstation clusters, through shared memory multiprocessors, to massively parallel distributed memory systems.

With code portability comes the opportunity for significant code re-use — commonly-used pieces of parallel code can be implemented once as libraries and re-used by a number of applications. Indeed, MPI was specifically designed to provide effective support for developers and users of parallel libraries. Many parallel applications have a similar structure, and libraries or abstractions encapsulating these structures can greatly simplify the task of developing a parallel implementation of an application. This idea of algorithmic skeletons was introduced in [Cole, 1989] and has been pursued more recently by a number of groups [Darlington *et al.*, 1993; MacDonald and Fletcher, 1994; Bruce *et al.*, 1995].

1.3 Performance Metrics for Parallel Systems

We can think of a parallel system as the combination of an algorithm and the platform on which it is executed (Figure 1–1). The challenge facing the system designer is to combine a particular algorithm with a particular platform to produce the "best" overall system design.

The "best" system is typically the one which achieves the highest performance. The performance of a system will depend upon the operations which the algorithm requires the platform to perform, and the performance achieved by the platform in carrying out these operations. An algorithm's behaviour, and therefore the sequence of operations it requests of the platform, is usually dependent upon its input. Hence the performance of a system can only be considered in the context of a particular input workload (Figure 1–2).

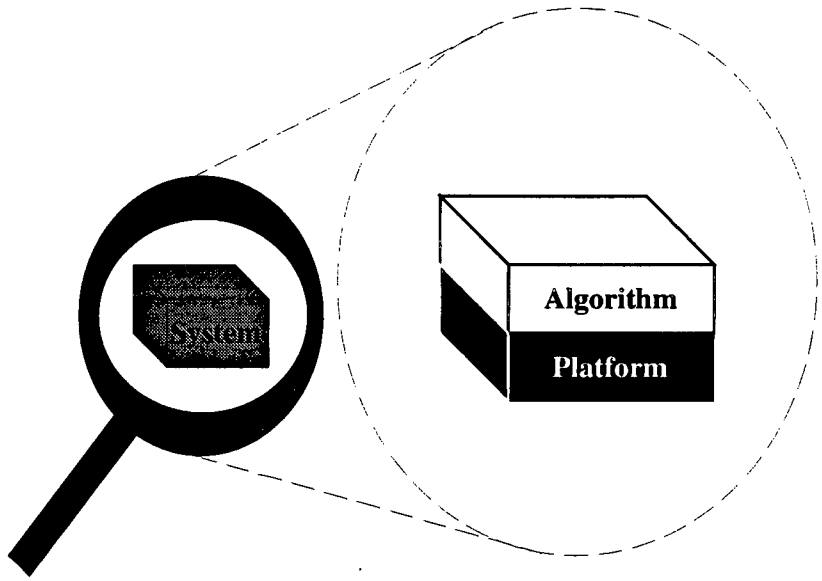


Figure 1–1: System = Algorithm + Platform

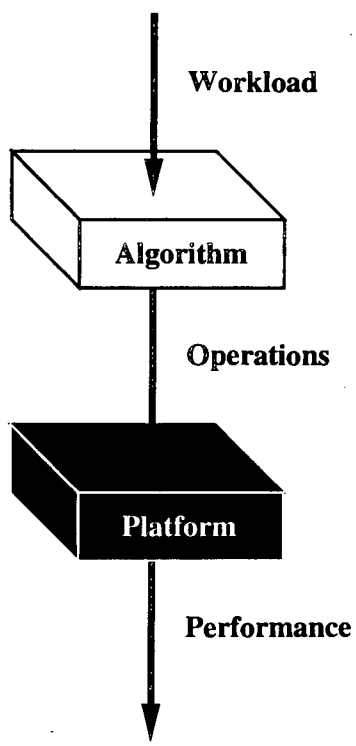


Figure 1–2: Performance is a function of workload, algorithm and platform

The performance of a system S under a workload W will depend upon attributes of S and W, and can be considered a function:

$$\text{Performance}\ (P_S; P_W) = \ldots \tag{1.1}$$

where P_S and P_W denote parameter sets characterising the system and the workload. We have defined a system to consist of an algorithm A running on a platform P. Hence the parameter set $P_S = P_A \bigcup P_P$, where P_A and P_P are parameter sets describing the algorithm and the platform respectively. We will write $(P_A; P_P)$ in place of P_S when it is convenient to distinguish between parameters in P_A and in P_P, and write Equation 1.1 as:

$$\text{Performance}\ ((P_A; P_P); P_W) = \ldots \tag{1.2}$$

In this section we will consider a number of performance metrics which a system designer can use to compare alternative designs. We are primarily interested in a system's performance in terms of the time the system takes to execute a particular workload. We can measure this as a time, as a rate, or use various dimensionless metrics, which are considered below.

1.3.1 Time

Execution time is perhaps the most commonly used performance metric. Let us denote the execution time of a system S under a workload W by

$$T_{\text{exec}}(P_S; P_W) \tag{1.3}$$

The execution times of two systems under the same workload can be directly compared to choose the system with the lower execution time.

1.3.2 Rate Metrics

Numerous metrics based on execution rates have been proposed. These all involve counting the number of "operations" of some sort carried out by the system and dividing this by the execution time. Let us denote the number of operations carried out by a system S under a workload W by

$$N_{\text{ops}}(P_S; P_W) \tag{1.4}$$

We can now define the execution rate of the system for that workload:

$$Rate(P_S; P_W) = \frac{N_{\text{ops}}(P_S; P_W)}{T_{\text{exec}}(P_S, P_W)} \tag{1.5}$$

If $N_{\text{ops}}(P_S; P_W)$ depends only on P_W, so that, for any S_1 and S_2,

$$N_{\text{ops}}(P_{S_1}; P_W) = N_{\text{ops}}(P_{S_2}; P_W) \tag{1.6}$$

then it is meaningful to compare $Rate(P_{S_1}; P_W)$ and $Rate(P_{S_2}; P_W)$ and select the system with the higher rate. For example, consider two transaction processing system designs S_1 and S_2 under a workload W consisting of n transactions. Taking $N_{\text{ops}}(P_{S_1}; P_W) = N_{\text{ops}}(P_{S_2}; P_W) = n$, we can sensibly compare $Rate(P_{S_1}; P_W)$ and $Rate(P_{S_2}; P_W)$.

Rate metrics can be abused in circumstances where $N_{\text{ops}}(P_S; P_W)$ depends upon the system parameters P_S as well as the workload parameters W_S. Common examples would be millions of instructions per second (MIPS) and floating-point operations per second (flop/s). A system might execute far fewer operations than another system with a simpler processor which must implement complex floating point operations in terms of many simpler operations. An analogous argument can be made in terms of CISC and RISC processors performing different numbers of instructions. In these and other such

System S	$N_{\text{operations}}(P_S, P_W)$	$T_{\text{execution}}(P_S, P_W)$	$Rate(P_S, P_W)$
S_1	300	25	12
S_2	100	10	10

Table 1–1: Inconsistencies between T_{exec} and $Rate$

cases, Equation 1.6 does not hold, and the rate obtained by two different systems cannot always be directly compared. Let

$$N_{\text{ops}}(P_{S_1}; P_W) = k_1 N_{\text{ops}}(P_{S_2}; P_W) \tag{1.7}$$

and

$$T_{\text{exec}}(P_{S_1}; P_W) = k_2 T_{\text{exec}}(P_{S_2}; P_W) \tag{1.8}$$

It follows that

$$Rate(P_{S_1}; P_W) = \frac{k_1}{k_2} Rate(P_{S_2}; P_W) \tag{1.9}$$

Note that it is possible for $k_1 > k_2$ to hold, suggesting that S_1 is the better design, while $T_{\text{exec}}(P_{S_1}; P_W) > T_{\text{exec}}(P_{S_2}; P_W)$, suggesting that S_2 is the better design. An example of such a contradiction is given in Table 1–1.

Using rate metrics to predict performance is also unreliable in general. The time taken to process a given workload on a particular system will typically depend on the particular sequence of operations involved as well as the total number. Even if

$$N_{\text{ops}}(P_S; P_{W_1}) = N_{\text{ops}}(P_S; P_{W_2}) \tag{1.10}$$

there is no guarantee that

$$T_{\text{exec}}(P_S; P_{W_1}) = T_{\text{exec}}(P_S; P_{W_2}) \tag{1.11}$$

nor therefore that

$$Rate(P_S; P_{W_1}) = Rate(P_S; P_{W_2}) \tag{1.12}$$

Super-scalar features and cache hierarchies are examples of system features which lead to context-specific execution times for operations on microprocessors. The time taken to execute a floating point operation will depend on which other floating point operations are in progress when the operation is issued; the time taken to fetch an operand from memory will depend on previous memory reference patterns.

Any rate metric, including flop/s or instruction/s, can be used to compare the performance of two systems, provided that N_{ops} depends only on the input workload, and not on the system.

1.3.3 Dimensionless Metrics

Speedup

We can compare the performance of two systems S_{base} and S under a workload W by taking a ratio which gives the execution time of S in terms of the execution time of the baseline system S_{base}:

$$\text{Ratio}(P_{S_{\text{base}}}; P_S; P_W) = \frac{T_{\text{exec}}(P_{S_{\text{base}}}; P_W)}{T_{\text{exec}}(P_S; P_W)} \tag{1.13}$$

In the case where S is a parallel system, the designer is often interested in how

$$\text{Ratio}(P_{S_{\text{base}}}; P_S; P_W)$$

changes with the number of processors used. Assuming that $p \in P_S$ represents the number of processors used, we can define:

$$\text{Speedup}(P_{S_{\text{base}}}; P_S; P_W; n) = \text{Ratio}(P_{S_{\text{base}}}; P_S|_{p=n}; P_W) \tag{1.14}$$

where $P_S|_{p=n}$ represents the parameter set P_S with p constrained to be n. This definition is known as "absolute" speedup. The absolute speedups of two systems S_1 and S_2, measured with respect to the same baseline system S_{base} can be directly compared — the system with the higher speedup has the higher performance. It is not always sensible to directly compare $\text{Speedup}(P_{S_{\text{base}}}; P_{S_1}; P_W; n)$ and $\text{Speedup}(P_{S_{\text{base}}}; P_{S_2}; P_W; n)$ for some specific n; a more interesting comparison would be of

$$\max_{n \in l \ldots u}(\text{Speedup}(P_{S_{\text{base}}}; P_{S_{\text{par}}}; P_W; n))$$

with $P_{S_{\text{par}}} = P_{S_1}$ and with $P_{S_{\text{par}}} = P_{S_2}$ for some interesting range of n bounded by l and u.

By taking $P_{S_{\text{base}}} = P_S|_{p=1}$ we obtain a definition of "relative speedup", or "self-speedup":

$$\text{Speedup}_{\text{rel}}(P_S; P_W; n) = \text{Speedup}(P_S|_{p=1}; P_S; P_W; n) \tag{1.15}$$

Relative speedups cannot be compared: a system with a higher speedup may in fact have the higher execution time and therefore lower performance.

Both $\text{Speedup}_{\text{rel}}$ and Speedup require the workload W to be the same, irrespective of the number of processors used. In many applications, it makes sense to scale the problem size with the number of processors used. This leads to an alternative definition of "scaled" speedup [Gustafson, 1988] (Equation 1.16).

$$\text{Speedup}_{\text{scaled}}(P_{S_{\text{base}}}; P_S; P_W; n; w) = \text{Speedup}(P_{S_{\text{base}}}; P_S; w(P_W; n); n) \tag{1.16}$$

where $w(P_W; n)$ scales the workload W for n processors.

Efficiency

Efficiency (Equation 1.17) is a measure of the effective utilisation of processors.

$$\text{Efficiency}(P_{S_{\text{base}}}; P_S; P_W; n) = \frac{\text{Speedup}(P_{S_{\text{base}}}; P_S; P_W; n)}{n} \tag{1.17}$$

The efficiency of two systems can be compared provided that the same base system is used. Efficiency generally falls as n increases for a fixed size workload W, and increases for fixed n as W increases. The trade-off between efficiency and speedup is explored in [Eager *et al.*, 1989].

Iso-efficiency

Given a function for the efficiency of a system under a particular workload, we can derive an iso-efficiency function [Kumar and Rao, 1987] (Equation 1.18). Such a function shows how fast the workload must be scaled with respect to the number of processors in order to maintain a certain level of efficiency, E.

$$\text{Isoefficiency}(P_{S_{\text{base}}}; P_S; P_W; E) = f \tag{1.18}$$

where, $\forall n_1, n_2$,

$$\text{Efficiency}(P_{S_{\text{base}}}; P_S; f(P_W; E; n_1); n_1) = \text{Efficiency}(P_{S_{\text{base}}}; P_S; f(P_W; E; n_2); n_2) \tag{1.19}$$

1.3.4 Cost-Performance

In some circumstances, there may be a performance threshold, with systems which achieve the threshold being assessed on the basis of some secondary metric, such as cost. If we neglect the cost of software development, we can consider the cost of a system to be a function C of the platform parameters P_P. If we are considering existing platforms, C is likely to be a partial function since not all combinations of platform parameters will be available. However, if we are interested in designing platforms, we may define a total function C and seek to optimise the cost performance given by Performance$((P_A; P_P); P_W)/C(P_P)$. A model of performance is clearly a pre-requisite for a model of cost-performance, and we will concentrate on models of performance.

1.4 Evaluating the Performance of Parallel Applications

Ideally, a performance evaluation methodology for parallel applications should:

1. allow the developer to predict the performance of hypothetical applications running on hypothetical platforms;

2. use natural and easily prepared characterisations of applications and platforms;

3. provide a symbolic expression for the performance of the system;

4. provide automated analysis techniques for deriving such symbolic expressions.

The simplest approach to performance evaluation involves constructing the implementation of the application, running it on the platform configuration in question, and measuring the performance directly. This is clearly not a predictive methodology, and does not provide a symbolic performance model. Furthermore, it requires access to the specific platform configuration in question. The need for a full implementation of the application makes this approach unsuitable as a design tool.

Rather than using real platform configurations, we can construct simulations of hypothetical platforms, and simulate the execution of our real applications on them. Proteus [Brewer *et al.*, 1992] is an example of an execution-driven simulator for parallel computer architectures. Although, like other simulators, Proteus is automated, it requires a simulation of the hypothetical platform of interest, as well as implementations of the application, and it does not provide a symbolic model of performance. To explore the parameter space of implementations or platforms, we must run a number of simulations. The biggest drawback of simulation is the large amount of simulation time required. Detailed simulations are so time consuming to run that parallel computers are being used to run them. The Wisconsin Wind Tunnel [Reinhardt *et al.*, 1993], for example, is an execution-driven discrete event simulation of cache coherent shared memory computers which runs on a massively parallel platform.

Another approach involves constructing executable models of application implementations, and executing them on the platform configurations of interest. Synthetic benchmarks seek to capture the structure of parallel applications, without performing the actual computation in detail. The ALPES project [Kitajima *et al.*, 1993; Tron *et al.*, 1994] and the LOOP language [Brehm, 1995] both use a library of high level operations which a programmer can call from a C program. An instrumented parallel program is automatically generated from this code, and executed on a parallel system. These tools do not provide us with a symbolic performance model, and exploring the parameter space of possible implementations involves running several analyses on different synthetic programs.

Many methodologies construct models of platforms and applications, and combine them to produce performance predictions. This is the most promising class of approaches for our purposes. At best, these methodologies use simple formalisms, allow us to explore hypothetical platforms and implementations, and perform automated analysis. A wide range of symbolic performance models have been proposed for various classes of applications and platforms, based on parametric characterisations. Other approaches use simulation techniques to derive symbolic expressions from parameterised executable models. A large number of methodologies provide formalisms for describing platforms

and applications, and analysis methods for deriving performance predictions from them. We will consider examples of all of these approaches below, and assess the extent to which they meet our criteria for a performance evaluation methodology.

1.4.1 Benchmarking

Benchmark suites such as SPEC [Dixit, 1991] are widely used to produce figures of merit for general workloads. Other suites assess system performance in more specialised application domains: the Livermore Loops [McMahon, 1986; Feo, 1988] and LINPACK [Dongarra, 1990] have been used to compare the performance of platforms on scientific workloads while database platforms typically report TPC benchmark figures [Transaction Processing Performance Council, 1994]. The widespread availability of portable application programming interfaces (APIs) on parallel systems has enabled the development of parallel benchmarks, such as GENESIS [Hey, 1991] and PARKBENCH [Hockney and Berry, 1994] to complement the PERFECT [Berry *et al.*, 1989] and SPLASH [Singh *et al.*, 1991] benchmark suites for multiprocessors.

By benchmarking the platforms of interest, and measuring the performance of our application on at least one of them, we could extrapolate to other platforms. In general, however, it is difficult to obtain accurate predictions by extrapolating from the performance of a benchmark to the performance of a particular implementation of an application. Any benchmark represents a particular workload and does not characterise the performance of a platform for other workloads. LINPACK is dominated by SAXPY or DAXPY calculations, which are typically amongst the highest performing operation sequences on modern microprocessors. It also has a regular, stride-1 memory access pattern, which is the most efficient pattern for using the memory hierarchy. As a result, any predictions based on LINPACK are likely to be over-optimistic. The Livermore Loops are dated in their programming style and do not include I/O, but the main concern is that they are vectorisable, memory-intensive loops, which may not be representative of whole applications.

Nor is it straightforward to extrapolate to the performance of a benchmark on a different configuration of the same platform, although some success in cross-platform prediction is reported in [Lin and Snyder, 1992]. Any such extrapolation will, of course, require some sort of modelling.

1.4.2 Direct algebraic models

A variety of manually-derived symbolic performance models for parallel applications have been proposed. Some of these are abstract models, applicable to any application or platform. Others are restricted to particular classes of platform or application, while some are specific to a particular application on a particular platform.

Platform-independent and Application-independent

A very familiar set of symbolic performance models can be derived from simple characterisations of workload and platform. We can characterise a workload W by the amount of work to be done w, and the parallelisable proportion of the work $0 \leq \alpha \leq 1$.

$$P_W = \{w, \alpha\} \tag{1.20}$$

We can characterise a sequential system S_{seq} by the rate r_{seq} at which its processor performs work.

$$P_{S_{\text{seq}}} = \{r_{\text{seq}}\} \tag{1.21}$$

We can now derive an expression for the execution time of a workload W on the system S_{seq}:

$$T_{\text{exec}}(P_{S_{\text{seq}}}; P_W) = \frac{w}{r_{\text{seq}}} \tag{1.22}$$

A parallel system S_{par} can be described by the number of processors p and the rate at which each processor performs work r_{par}.

$$P_{S_{\text{par}}} = \{p, r_{\text{par}}\} \tag{1.23}$$

Assuming that the parallelisable work is perfectly load-balanced across the available processors, and that any necessary communication and synchronisation is performed instantaneously by the platform, we can derive an expression for the execution time of the workload W on the system S_{par}.

$$T_{\text{exec}}(P_{S_{\text{par}}}; W) = \frac{w((1-\alpha) + (\alpha/p))}{r_{\text{par}}} \tag{1.24}$$

We can derive expressions for relative speedup, absolute speedup, and efficiency.

$$\text{Speedup}_{\text{rel}}(P_{S_{\text{par}}}; P_W; n) = \frac{1}{(1-\alpha) + (\alpha/n)} \tag{1.25}$$

$$\text{Speedup}(P_{S_{\text{seq}}}; P_{S_{\text{par}}}; P_W; n) = \frac{r_{\text{par}}}{r_{\text{seq}}((1-\alpha) + (\alpha/n))} \tag{1.26}$$

$$\text{Efficiency}(P_{S_{\text{seq}}}; P_{S_{\text{par}}}; P_W; n) = \frac{1}{n(1-\alpha) + \alpha} \tag{1.27}$$

[Driscoll and Daasch, 1995] allow α to be a linear or logarithmic function of p rather than a constant. This allows the sequential fraction of the execution time to rise or fall as the number of processors increases.

In [Moncrieff *et al.*, 1995], this model is used to relate two parallel systems characterised by $P_{S_1} = \{p_1, r_1\}$ and $P_{S_2} = \{p_2, r_2\}$, deriving an expression for α_{critical}, the minimum parallelisable proportion required for one system to be able to out-perform the other (Equation 1.28).

$$\alpha_{\text{critical}} = \frac{1}{1 - \left(\frac{1}{p_1 r_1} - \frac{1}{p_2 r_2}\right) / \left(\frac{1}{r_1} - \frac{1}{r_2}\right)} \tag{1.28}$$

Although attractively simple, these models have a number of limitations. Accurately determining w and α is difficult, and may require profiling of an implementation. The rate of execution r depends heavily on the program being executed. Finally, the definition of T_{exec} assumes that load balance is perfect, and communication is instantaneous and without overhead. Even if accurate values of w, α and r can be obtained, the model produces unrealistically optimistic predictions.

Hockney takes a more detailed approach, introducing a concept of operations, and deriving expressions for execution time from the asymptotic rate r_∞ at which a particular type of operation can be performed, and the amount of work $n_{1/2}$ required to achieve half this level of performance. By characterising applications as a series of operations of given sizes, an expression for execution time can be developed [Hockney, 1987; Hockney and Curington, 1989; Hockney, 1991]. Although originally used to characterise vector performance, this approach has also been applied to memory and communication operations.

Hockney characterises a platform in terms of 5 parameters:

- r_∞^s : the asymptotic rate at which scalar computations are performed
- r_∞^v : the asymptotic rate at which vector operations are performed
- r_∞^c : the asymptotic rate at which data is communicated between processors
- $n_{1/2}^v$: the length of vector required to achieve half the asymptotic vector performance
- $n_{1/2}^c$: the length of message required to achieve half the asymptotic message-passing performance

and characterises an application in terms of a further 5 parameters:

- s^s - the total amount of scalar work to be done
- s^v - the total amount of vector work to be done
- s^c - the total amount of communications work to be done
- n^v - the average vector length
- n^c - the average message length

Platform models can be calibrated experimentally, and applications characteristics can be provided by the designer or derived from profiling. The larger number of parameters in Hockney's model increases the amount of calibration and characterisation which may be done, but the differentiation of different types of operations should allow more accurate predictions. Measuring $n_{1/2}$ for a given operation will depend significantly on the memory hierarchy and the stride used. Also, modelling all scalar processing as a single operation and all vector processing as an operation is a extreme simplification, which does not address the very wide variation in performance which would be expected between different operations in these classes. This makes Hockney's approach dependent on the mix of operations in a given program.

Given these characteristics, Hockney produces a model of execution time given by:

$$\begin{aligned} T_{\text{par}} &= \frac{s^s}{r^s_\infty} \\ &+ \frac{s^v}{r^v_\infty} + \frac{\frac{s^v}{n^v} n^v_{1/2}}{r^v_\infty} \\ &+ \frac{s^c}{r^c_\infty} + \frac{\frac{s^c}{n^c} n^c_{1/2}}{r^c_\infty} \end{aligned}$$

Symbolic suitability functions measuring the fit of application and architecture can also be derived [Getov *et al.*, 1993; Getov and Hockney, 1993]. The function $\alpha^{n/v}$ (Equa-

tion 1.29) represents the suitability of the application's vectorised fraction to the vector/scalar ratio of the system. This function is always greater than one and ideally should be as close to one as possible in order to make efficient use of the vector unit.

$$\alpha^{n/v}(N,p) = \frac{r_\infty^n}{r_\infty^v} \frac{s^v(N,p)}{s^n(N,p)} \tag{1.29}$$

$\alpha^{n/c}$ (Equation 1.30) measures the fit of the application to the architecture in terms of calculation/communication ratio. The best fit (not the best performance) is when $\alpha^{n/c} = 1$. If $\alpha^{n/c} \ll 1$, then the communications costs are very low and high performance is achieved. If $\alpha^{n/c} \gg 1$ then the performance dramatically decreases because lots of time is being spent in very intensive communications.

$$\alpha^{n/c}(N,p) = \frac{r_\infty^n}{r_\infty^c} \frac{s^c(N,p)}{s^n(N,p)} \tag{1.30}$$

Hockney's model assumes that scalar, vector and communications work cannot be overlapped, and therefore provides a worst-case prediction. In reality, architectural features such as direct memory access for communication, vector chaining, pipelining and superscalar execution all achieve some level of overlapping leader to lower execution times.

Platform-specific and Application-specific

A typical example of this approach is described in [Zemerly *et al.*, 1995], which performs "bottleneck analysis" of a parallel linear solver by developing symbolic expressions for various components of the execution time. Bottlenecks are deemed to occur when ratios of symbolic expressions, such as the ratio of communication time to computation time, or the ratio of memory access time to processing time, are greater than one. This form of bottleneck analysis is flawed, since it takes no account of bottlenecks which occur from a combination of resources. In practice, the critical path for a parallel application's execution often spans more than a single type of resource. The main

problem with this approach, however, is the ad hoc nature of the modelling process. Each developer must construct an expression for the execution time of each application on each platform, and unrealistic assumptions are often made in order to produce comparatively simple expressions.

Classes of platforms, classes of applications

A number of researchers have developed parameterised symbolic models of execution time for various common parallel application structures. By characterising a particular application and platform, developers can produce performance models for a given application in a class, re-using the generic model which has been developed for that class. Like application-specific models, this type of model often makes unrealistic assumptions in order to produce comparatively simple expressions.

Algebraic expressions for the execution time of iterative algorithms on multiprocessors are presented in [Vrsalovic *et al.*, 1988]. Each iteration consists of some amount of access to global data and some amount of local processing. Lower and upper bounds on execution time are developed by assuming no contention and maximal contention respectively. The effects on performance of processor, memory and interconnection characteristics are explored, and speedup is evaluated symbolically. However, the models assume perfect load balance and take no account of inherently serial aspects of the computation. Within these constraints, a developer could predict the performance of an iterative algorithm by providing values for the parameters which characterise the platform and the application.

The performance of parallel implementations of explicit solvers for partial difference equations is characterised algebraically in [Reed *et al.*, 1987]. Although a variety of multiprocessor and multicomputer architectural models are studied, this work takes no account of contention in the network. Again, a developer could exploit these models to make predictions, given parameter values for the characterisations of the platform and the application.

A variety of parallel application structures are studied in [Sussman, 1991]. Symbolic expressions are developed for various components of execution time, and these are then combined to provide a performance prediction model. This prediction assumes that no overlapping between various components of execution time is possible, and hence represents an upper bound on performance prediction. Sussman's work attempts to address the potential for overlapping components of execution time. Unfortunately, this is not derived from the application and platform descriptions during analysis, but must be provided as a parameter by the developer. The amount of overlapping is typically a complex function of the application and the platform, and it is unreasonable to expect the developer to provide a value for it. Moreover, using a single explicit parameter value is dubious, since the level of overlapping will vary in different parts of the application depending on dynamic execution patterns.

Performance models based on skeletons and application classes can be useful, but they are limiting, since they do not easily allow the programmer to investigate the performance of slightly different parallel application structures. Moreover, skeleton performance models are often parameterised on characterisations which can be difficult for the developer to provide manually. However, given characterisations of platform and application, the developer can automatically produce a performance model.

1.4.3 Simulation approaches

Simulation approaches typically allow the developer to investigate performance at a particular point in parameter space. To determine performance for another set of parameter values, a separate simulation must be run.

One technique for constructing a symbolic performance model from simulation results is to conduct factorial experiments, systematically varying parameters of a synthetic program. Rigorous analysis of the interactions between parameters can then be performed, and a model fitted. An example of this approach is reported in [Candlin *et al.*, 1992], which also presents accurate predictions obtained by interpolation in parameter space. The technique assumes a linear interpolation, which does not account for non-

linear interactions between parameters. The accuracy of linear extrapolations to other regions of parameter space is not clear.

1.4.4 Formal approaches

Formal approaches provide a formal notation which is used to describe the platform and the application, and an analysis method which takes these formal descriptions and produces performance predictions. Formal approaches allow models to be handled more reliably than could be done manually. Where automatic analysis techniques are available, larger models can typically be analysed.

Complexity theory

Complexity analysis techniques formalise the modelling assumptions made in deriving symbolic expressions for execution time. The PRAM [Fortune and Wyllie, 1978] is an idealised model of a parallel machine with synchronous processors and a shared memory. The unrealistic uniform memory access costs of the PRAM led to the development of the Block PRAM (BPRAM) [Aggrawal *et al.*, 1989] in which memory access costs depend on the amount of data transferred. Other models incorporating more realistic cost models include the Module Parallel Computer [Alt *et al.*, 1987], BSP [Valiant, 1990], LogP [Culler *et al.*, 1993], the HPRAM [Heywood and Ranka, 1992], and CLUMPS [Campbell and Turner, 1994].

Unfortunately, the results of complexity analyses which use these machine models are typically reported as asymptotic bounds. Consider the execution times T_1 and T_2 of two systems S_1 and S_2 respectively, given in Equations 1.31 and 1.32.

$$T_1(n) = c_1 n^2 \tag{1.31}$$

$$T_2(n) = c_2 n \tag{1.32}$$

Asymptotic analysis would conclude that S_2 achieved higher performance, irrespective of the values of c_1 and c_2. If $c_2 = 100$ and $c_1 = 2$ then S_2 only achieves higher performance for $n > 25$, which may not be the regime of practical interest. Moreover, asymptotic analysis would be unable to distinguish two systems with execution time $100n^2$ and $4n^2$. While some parameters such as problem size may have large values dwarfing the constant factors associated with them, other parameters may be much closer in magnitude to the constants. This is particularly true of parallel systems, since the number of processors p will typically not grow to very large values.

Micro-analysis techniques

Micro-analysis approaches develop symbolic expressions, parameterised on platform characteristics, for the execution time of programs. Micro-analysis approaches have been applied to sequential Pascal and C programs [Cohen and Weitzman, 1992] and a sequential molecular dynamics kernel [de Ronde *et al.*, 1994].

A good example of micro-analysis of parallel applications is [Hickey *et al.*, 1992], which develops expressions for the execution time of Ada programs. A machine-independent representation of a program's execution is constructed from the program and its input data. This is then combined with a machine description to produce an execution trace, from which a symbolic expression is derived. Varying the number of processors requires a new trace to be generated, although the machine-independent representation of the execution can be re-used.

Much work with micro-analysis has been in relatively benign environments without aggressive compiler optimisation and high performance processor architectures.

Graph Models

Directed graphs are commonly used as models of programs, capturing the dependencies between different components of a calculation. A wealth of literature tackles the issue of how programs represented by such graphs can be scheduled onto parallel computers,

which are typically characterised by a small number of scalar parameters. An excellent survey of this field is presented in [Norman and Thanisch, 1991]. Many scheduling models for directed graphs make grossly unrealistic assumptions about communication costs on parallel computers. With an appropriate cost model, which takes account of bandwidth and message startup time, these models are useful for deriving a lower bound on parallel execution time, evaluating the extent to which intertask dependencies constrain parallelism. However, models of this type struggle to take account of the resource contention which may be implied by the communication patterns, and which may significantly effect the execution time. This makes them particularly unsuitable for bus networks, a commonly used architecture in workstation clusters.

Examples of this approach, which all suffer from the weaknesses discussed above, are [Brehm *et al.*, 1995], [Mendes, 1993] and [Zhang and Xu, 1995]. [Brehm *et al.*, 1995] models parallel applications as task graphs in which each node is labelled with a symbolic expression for the number of floating point operations it performs. The computational performance of a node is obtained by measuring the sustained execution rate of a sequential version of the application. Communication performance is characterised by generic benchmarks. These platform characteristics are used to derive a prediction from the task graph. [Mendes, 1993] describes a performance prediction technique based on trace transformation. An event graph is derived from an execution trace produced by running an instrumented parallel application. Predictions for other platforms are obtained by transforming the event graph. Machine characteristics, obtained from benchmarks, are used to scale the duration of intervals between events, taking care to preserve causality relationships between events on different processors. [Zhang and Xu, 1995] uses a "thread graph", an enhanced task graph that includes explicit communication dependencies and operations, to explore the scalability of parallel applications.

A number of other approaches rely on stochastic models. The ES methodology [Sinclair and Dawkins, 1994] describes applications as task systems — task graphs in which each task is labelled with a random variable denoting its execution time, and a list of the resources required by that task. A task system represents a set of different execution sequences, and the ES approach enumerates these possible orderings to determine an

average execution time. The combinatorial explosion which results is tackled, to some extent, by pruning heuristics which reduce the number of sequences which must be considered. However, the combinatorial explosion remains a problem. [Malony *et al.*, 1994] describes a modelling language which can be used to describe stochastic graph models of workload. These can be analysed accurately using Markov analysis, but state space explosion precludes this technique for all except small problems. Bounding methods and approximate techniques such as series reduction and parallel reduction of the stochastic graph are presented. [Sötz, 1990] relaxes the series-parallel constraint and presents an approximate technique for determining the average runtime of a program with deterministically or exponentially distributed task durations.

The results of a recent study [Adve and Vernon, 1993] show that non-deterministic processing requirements and random communication delays introduce negligible variance into the execution time of shared memory programs. The authors argue that deterministic models are adequate for performance prediction of parallel systems and criticise the widespread use of exponential task times in stochastic analyses of parallel systems, presenting empirical evidence that process execution times approach a normal distribution.

Resource contention models

[Qin *et al.*, 1991] describes a shared memory model in which processes contend for objects in shared memory through locks. A combination of symbolic and simulation analysis techniques are implemented in a software tool (TCAS). This formalism cannot naturally describe message-passing programs, which are the prevalent form of parallel software.

PAMELA [van Gemund, 1993b; van Gemund, 1993c] is a much more mature formalism which provides abstract language for describing parallel applications in which processes contend for resources through semaphore operations. Restricted, but commonly occurring sub-classes of PAMELA programs are amenable to serialisation analysis, an algebraic approximation technique which produces upper and lower bounds on execution

time. In general, a discrete event simulation is used to simplify PAMELA models [van Gemund, 1993a].. This semaphore model is more readily applied to modelling other forms of contention, such as network contention due to message-passing. However, to model this, the developer must specify the resources used by his program, right down to the finest level required. For simple networks, such as a bus, this is straightforward. For complex interconnection networks, this requires the developer to explicitly request each link and switch resource required to transmit a message. PAMELA does not readily separate this sort of platform consideration from the application model. Nevertheless, some interesting results have been achieved, and serialisation analysis allows large models to be tackled.

Queueing Theory

Performance predictions can be derived efficiently from closed (separable) queueing networks using mean value analysis. Such models capture mutual exclusion (e.g. resource contention) well, but cannot capture condition synchronisation. Non-product form networks can address this problem, but such networks require more demanding analysis, through simulation. For example, a non-product form model of matrix multiplication on a multicomputer is described in [Smirni and Rosti, 1994].

A number of hybrid approaches, combining task graph models of applications and queueing network models of platforms, have been proposed. [Mak and Lundstrom, 1990] models computations as series-parallel directed acyclic graphs, and resources as service centres in a queueing model. A similar approach, which exploits symmetries and replications in the model to improve efficiency of analysis, is described in [Jonkers, 1994b]. Work reported in [Kapelnikov *et al.*, 1989; Kapelnikov *et al.*, 1992] uses a queueing network to describe the platform, and a "computation control graph" (CCG) to describe the application. The CCG is an extension of a directed acyclic graph which can conveniently represent recursion, looping constructs, multiple instantiations of tasks and hierarchical grouping of dependencies. A Markov process is generated from the two models, and a number of heuristic approximation techniques are used to tackle

large problems. All of these hybrid approaches require significant computational power for analysis, generate only numerical performance predictions and are quite limited in terms of the model sizes which are tractable. Moreover, the combination of multiple formalisms in an attempt to address the frailties of queueing theory for modelling parallelism is more challenging for the developer and less satisfactory than an integrated approach to the problem.

Petri Nets

Petri nets can express condition synchronisations which queueing networks cannot capture. Their major drawback is their analysis complexity, which is typically exponential in the size of the net. Analysis of stochastic Petri nets can be so demanding that parallel simulations of Petri net models have been proposed [Ferscha and Chiola, 1994; Caselli *et al.*, 1994].

As with queueing theory, a number of hybrid approaches have been proposed. A similar approach presented in [Childers *et al.*, 1995] combines a graph model of software, a model of hardware structure, and a mapping of the software graph on to the hardware, to produce a C program which represents a generalised stochastic Petri net model. [Jonkers, 1994a] combines queueing networks with Petri nets, yielding a model that can capture synchronisation behaviour, but which is somewhat more amenable to efficient analysis than a pure Petri net model. These approaches still require significant computational power during analysis, and, like other hybrid approaches, demand greater sophistication from the developer than a single integrated formalism.

Ferscha's PRM-Net methodology [Ferscha, 1992] is an elegant, and well-integrated use of a single formalism. Ferscha separates the program model from the platform model, using one Petri net to describe the program's control flow, another to describe the platform, and a mapping describing how the program is mapped onto the platform. These are combined to produce a single stochastic Petri net describing the whole system, which can be analysed using traditional numerical techniques. The PRM-Net methodology has been applied to substantial numerical applications on a range of platforms, and

a performance evaluation environment based on the PRM-Net methodology is being developed [Wabnig and Haring, 1994a; Wabnig and Haring, 1994b]. Despite its strengths, the PRM-Net methodology suffers from resource-hungry analysis and does not provide a symbolic performance model.

1.4.5 Conclusions

For the prediction of sequential execution time, micro-analysis is an interesting approach, but its effectiveness has not been assessed in the presence of optimising compilers and superscalar microprocessors which are common in the parallel computing environment. Little work has been done on extending micro-analysis to SPMD programs which use collective interprocessor communication, an important form of parallel application.

In order to analyse large models, we require an automated analysis technique. A formalism is clearly necessary for supporting automated analysis, but describing parallel applications in existing formalisms is often awkward and unnatural. Those approaches which can naturally describe parallel applications do not produce symbolic models of performance.

1.5 Synopsis

This thesis develops and evaluates performance prediction methods which can support the programmer during the development of parallel applications. In particular, we focus on message-passing programs written in Fortran 77 using MPI, executing on MIMD parallel systems based on commodity microprocessors. Our goal is to develop *predictive* performance evaluation techniques which apply *automatic* analysis to *natural* descriptions of systems and applications, producing *symbolic* performance models. The accuracy with which these models predict the performance of applications will also be assessed.

In order to predict the performance of a parallel application accurately, we must be able to predict the performance of its sequential components. We assume that we have an existing sequential implementation of the application from which the sequential components of the parallel application can be extracted. Micro-analysis techniques have been used to derive symbolic expressions for the execution time of programs written in a variety of languages. However the programming languages used in these studies are not usually associated with aggressive optimising compilers, while the hardware platforms used are not sophisticated modern microprocessors. In Chapter 2 we extend previous work on micro-analysis by assessing its effectiveness in the presence of aggressive compiler optimisation, on scientific workstation platforms. We present a micro-analysis technique which straightforwardly derives symbolic expressions for execution time from a subset of Fortran 77. Like some other micro-analysis approaches, this derivation could easily be automated. We use an automatic method to characterise the performance of several uni-processor platforms, and these characterisations are combined with the symbolic performance expressions to predict the execution time of a number of code fragments. This work extends previous work on micro-analysis by exploring the effect of aggressive compiler optimisation on the accuracy of predictions. Furthermore, we assess the accuracy with which the execution time of sequential code fragments can be predicted given perfect knowledge of control flow.

In Chapter 3 we move on to predict the performance of parallel applications. As discussed above, a number of researchers have developed accurate symbolic performance models for particular parallel application skeletons. However, these performance models are typically derived manually, and are useful only if the parallel application's structure complies precisely with the skeleton. We explore a more flexible technique which constructs symbolic performance models for parallel application structures from models of the collective communication operations which implement them. This is a natural extension of the micro-analysis technique applied to sequential code fragments in the previous chapter. We identify a number of commonly occurring parallel application structures, and describe how they can be implemented using collective communication operations. The performance of these collective communication operations is evaluated

on a number of parallel platforms and simple symbolic models are fitted to the measured data. The measured performance of skeletal parallel applications is compared with predictions derived symbolically and numerically from the measured performance of individual collective communication operations. This allows the accuracy with which communication performance can be predicted to be assessed.

In Chapter 4 we develop more detailed symbolic models of collective communication operations, using micro-analysis techniques. We combine these models to derive expressions for the execution time of a number of parallel applications. We demonstrate the use of symbolic performance models in the design process, to optimise a single implementation, to compare alternative implementations and to compare alternative platforms. Interaction between communication, in the form of network contention, has a major effect on parallel application performance. These interactions are difficult to reason about informally, and are the major source of complexity in deriving the symbolic expressions presented in this chapter. Manual derivation of these expressions is only practical for extreme scenarios in which contention is maximal (e.g. a bus network) or minimal (i.e. a perfect network). In reality, many interesting parallel platforms lie between these extremes. There is a clear need for an automatic analysis tool which could be used to check manually derived expressions, or automatically produce expressions for complex systems.

A number of formalisms, and their accompanying automatic analysis techniques, have been proposed for modelling parallel applications. Few of these provide concepts familiar to the parallel application developer, such as processes and communication. Indeed, some formalisms struggle to express or capture essential aspects of parallel computation, such as synchronisation between processes. Moreover, many of these formalisms are based on stochastic models, while deterministic models are arguably adequate in this context, and more accessible to the developer.

Process calculi, built on basic concepts of processes and communication, have been widely used to reason about behavioural properties of interacting concurrent systems. Timed extensions of process calculi are used to reason about time-dependent behavi-

oural properties. In Chapter 5, we define a timed process calculus, Eager Timed Calculus of Communicating Systems, which is specifically intended for reasoning about the performance of concurrent systems. We demonstrate Eager Timed CCS in use, discuss how the analysis of Eager Timed CCS models can be automated, and report on an automatic analysis tool for Eager Timed CCS which has been implemented in Standard ML.

Chapter 6 explores how processes, processors and parallel applications can be modelled in Eager Timed CCS. Models of parallel applications are analysed using the tool introduced in the previous chapter, and the performance and resource requirements of the tool are evaluated.

A limitation of Eager Timed CCS, as defined in Chapter 5 and implemented in the analysis tool, is the requirement that time variables have concrete values. As a result, an application's parameter space can only be explored by analysing a number of instances of a model defined by particular sets of parameter values, in much the same way as a simulation approach. Chapter 7 presents a parametric semantics for Eager Timed CCS, which allows models containing time variables to be analysed directly. This extension has important implications for automated analysis, necessitating sophisticated constraint-satisfaction algorithms. We describe a new analysis tool, implemented in Standard ML, which incorporates these algorithms, and demonstrate its use in the analysis of simple parametric systems. We close the chapter with an assessment of the tool's resource requirements and its applicability to the analysis of larger systems.

Finally, Chapter 8 summarises the contributions of this thesis, and discusses possible extensions and directions for further work.

Chapter 2

Predicting the Execution Time of Sequential Applications

In order to predict the execution time of a parallel program, there is clearly a need to predict the execution time of its sequential components. Any performance prediction methodology will involve characterising an application in terms of the workload it places on the system, and characterising the system's performance in terms of its workload. If we characterise an application by the count c of operations it requires performed, and characterise the platform by the rate r at which it can perform operations, we might predict the execution time of the system as $\frac{c}{r}$.

We could, for example, characterise scientific applications by the number of floating point operations which they require to be executed, and characterise platforms by the rate at which they can execute floating point operations. Platforms rarely approach their advertised peak execution rates, so an alternative measure is required. Since the execution time of a code will depend upon the instruction mix and the other operations executed, in reality a platform's floating-point performance will vary. The Livermore Loops [McMahon, 1986; Feo, 1988] are one commonly-used benchmark of this performance range. These are a set of 24 code fragments extracted from application codes, and a geometric mean of their performance can be calculated as a good central measure.

Table 2–1 and Figure 2–1 show that, even for the 24 Livermore Loops on which the measurements are made, a performance prediction on the basis of the geometric mean

Metric	*SUN 4/50*	*DEC 3300L*	*SGI Indigo2*
Maximum (Mflop/s)	4.06	4.09	19.82
Geometric Mean (Mflop/s)	1.51	1.52	7.17
Minimum (Mflop/s)	0.74	0.58	2.46
Maximum/Geometric	2.68	2.69	2.76
Minimum/Geometric	0.49	0.38	0.34

Table 2–1: Mflop/s rates from Livermore Loops with minimal compiler optimisation

can overestimate performance by more than 160% and underestimate performance by more than 50%. Hence this simplistic approach looks likely to produce predictions which are within a factor of three of the actual performance of a code fragment. Of course, the performance of other codes may well be predicted less accurately.

In search of more accurate predictions, we can generalise our approach, characterising an application by an n-tuple $(c_1, \ldots, c_n)$ of counts for operations of different types, and characterising a platform by the n-tuple $(r_1, \ldots, r_n)$ of rates at which it performs operations of these types. Equivalently, we can describe a platform by an n-tuple of times $(t_1, \ldots, t_n)$ corresponding to the execution time of each of the basic operations. We will then predict execution time to be $\sum_{i=1}^{n} \frac{c_i}{r_i}$, or alternatively $\sum_{i=1}^{n} c_i t_i$. This approach should increase the sensitivity of predictions to different operation mixes, and thereby improve their accuracy. Micro-analysis approaches of this type have been applied to estimating the execution times of Strassen's matrix multiplication algorithm, deterministic parsers, and a class of straight-line programs [Cohen, 1982], Pascal or C-like programs [Cohen and Weitzman, 1992] and a molecular dynamics kernel [de Ronde *et al.*, 1994].

In this chapter, we will evaluate the effectiveness of applying micro-analysis techniques to Fortran 77 code fragments, running on a variety of platforms based on RISC microprocessors. We would expect the sophisticated optimisations typically performed by Fortran 77 compilers, and the complexity of modern RISC processor architectures to provide additional challenges for the micro-analysis approach. To assess the accuracy

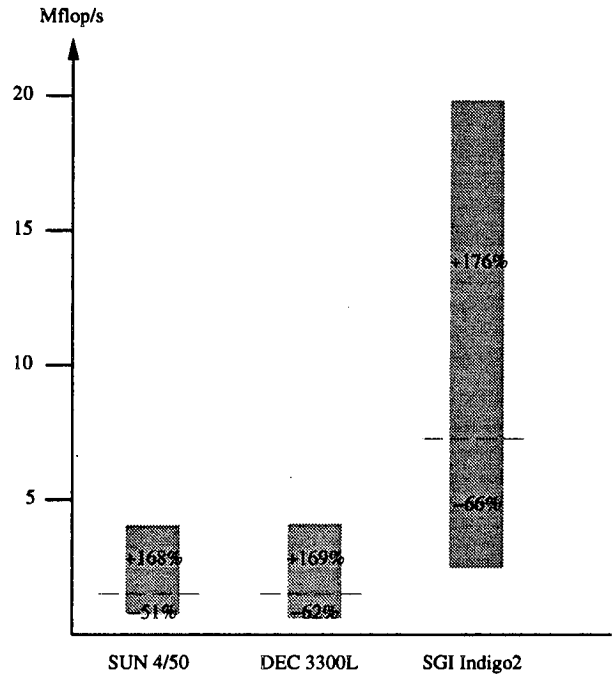


Figure 2–1: Mflop/s ranges from Livermore Loops with minimal compiler optimisation. Percentages shown are maximum and minimum relative to the geometric mean.

of our approach, we will compare our results with a simpler approach based on Mflop/s rates and flop counts.

2.1 A Programming Language and Cost Model

Our experiments will be based on a subset of Fortran 77, shown in Table 2–2. This language is completely deterministic; we assume that we have perfect knowledge about control flow, so we know exactly which statements will be executed. This is the best-case scenario for performance prediction. A cost model for micro-analysis comprises a tuple of operations $(o_1, \ldots, o_n)$, and a mapping M from the program language L to operation counts $(c_1, \ldots, c_n)$. We use the operations loopoh, load_I, store_I, add_I, mult_I, load_R, store_R, add_R and mult_R presented in Table 2–3 to characterise programs and platforms. The mapping in Table 2–4 makes a number of assumptions, notably that scalar variables will be held in registers and therefore do not incur load and store costs.

```
      ident  ::=  scalarident
               |  arrayident

scalarident  ::=  A|...|Z
 arrayident  ::=  scalarident(expr)

      expr  ::=  expr + expr
               |  expr - expr
               |  expr * expr
               |  expr / expr
               |  (expr)
               |  ident
               |  constant

   stmtlist  ::=  stmt
               |  stmt
                  stmtlist

      stmt  ::=  ident = expr
               |  DO scalarident = expr, expr
                   stmtlist
                  END DO
```

Table 2–2: Syntax of programs

Operation $\in O$	*Definition*
loopoh	Overhead of loop counter increment, conditional and branch
load_I	Integer load
store_I	Integer store
add_I	Integer addition or subtraction
mult_I	Integer multiplication
load_R	Floating point load
store_R	Floating point store
add_R	Floating point addition or subtraction
mult_R	Floating point multiplication

Table 2–3: Operation set and their intended meanings

2.2 Obtaining a Platform Model

To predict the execution time of codes on a particular platform given their time formulae, we must obtain measurements of the basic operation times on that platform. We do this by measuring the execution time of a number of code fragments and constructing a set of simultaneous equations in which the predicted execution time formulae, derived from the fragments using the mapping given above, are equated to the measured execution times. The unknowns in this set of equations are the basic operation times for the platform in question. The equations are very unlikely to admit an exact solution, since the model is an approximation. The restriction of linearity which we place on time formulae allows us to use a least squares or multiple linear regression technique to find an approximate solution.

A number of difficulties must be addressed in selecting the set of experiments from which to generate the set of simultaneous equations. We must conduct at least as many experiments as there are basic operations, and in general, the larger the set of experiments the better. Unfortunately, some basic operations will appear disproportionately often in many sets of experiments, and this can lead to a very poorly conditioned matrix

$$
\begin{aligned}
T_{\text{expr}}[\![\text{expr}_1 + \text{expr}_2]\!] &= T_{\text{expr}}[\![\text{expr}_1]\!] + t_{\text{add}_r} \\
&+ T_{\text{expr}}[\![\text{expr}_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
T_{\text{expr}}[\![\text{expr}_1 \text{ - } \text{expr}_2]\!] &= T_{\text{expr}}[\![\text{expr}_1]\!] + t_{\text{add}_r} \\
&+ T_{\text{expr}}[\![\text{expr}_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
T_{\text{expr}}[\![\text{expr}_1 * \text{expr}_2]\!] &= T_{\text{expr}}[\![\text{expr}_1]\!] + t_{\text{mult}_r} \\
&+ T_{\text{expr}}[\![\text{expr}_2]\!]
\end{aligned}
$$

$$
T_{\text{expr}}[\![\text{(expr)}]\!] = T_{\text{expr}}[\![\text{expr}]\!]
$$

$$
T_{\text{expr}}[\![\text{constant}]\!] = 0
$$

$$
T_{\text{expr}}[\![\text{scalarident}]\!] = 0
$$

$$
T_{\text{expr}}[\![\text{scalarident (expr)}]\!] = t_{\text{load}_r}
$$

$$
T_{\text{stmt}}[\![\text{ident} = \text{expr}]\!] = T_{\text{expr}}[\![\text{expr}]\!]
$$

$$
T_{\text{stmt}}[\![\text{scalarident (expr}_1\text{)} = \text{expr}_2]\!] = t_{\text{store}_r} + T_{\text{expr}}[\![\text{expr}_2]\!]
$$

$$
\begin{aligned}
T_{\text{stmt}}\left[\!\!\left[\begin{array}{l} \texttt{DO S} = \text{E}_1, \text{E}_2 \\ \quad \text{stmtlist} \\ \texttt{END DO} \end{array}\right]\!\!\right] &= (\text{E}_2 - \text{E}_1 + 1) \\
&\times (t_{\text{loopoh}} + T_{\text{stmtlist}}[\![\text{stmtlist}]\!])
\end{aligned}
$$

$$
T_{\text{stmtlist}}[\![\text{stmt}]\!] = T_{\text{stmt}}[\![\text{stmt}]\!]
$$

$$
\begin{aligned}
T_{\text{stmtlist}}\left[\!\!\left[\begin{array}{l} \text{stmt} \\ \text{stmtlist} \end{array}\right]\!\!\right] &= T_{\text{stmt}}[\![\text{stmt}]\!] \\
&+ T_{\text{stmtlist}}[\![\text{stmtlist}]\!]
\end{aligned}
$$

Table 2–4: Mapping from programs to time formulae

of coefficients. Care must also be taken to avoid equations which are almost linearly dependent, since the small differences in basic operation coefficients can lead to a set of equations with a very unstable fit. Equation 2.1 presents the coefficients used to determine the basic operation times used in this chapter.

$$\begin{pmatrix} \text{loopoh} & \text{load}_I & \text{store}_I & \text{add}_I & \text{mult}_I & \text{load}_R & \text{store}_R & \text{add}_R & \text{mult}_R \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Several further issues arise in timing the experiments. In addition to ensuring that the platform is dedicated to the experiment when measurements are made, account must also be taken of the clock resolution. The high clock period on some platforms typ-

ically means that a code fragment must be executed many times in order to eliminate significant quantisation errors. It is easy to implement a wrapper which increases the number of iterations and repeats the experiment until the elapsed time is large with respect to the clock period. Of course, this can cause problems since cache miss costs may well only be incurred during the first iteration of the algorithm. Finally, the time taken to read the clock can be significant relative to the execution time being measured. It is important to determine the time taken to read the clock and to normalise the measured execution times of experiments accordingly if necessary.

Table 2–5 gives basic operation times obtained using this method on a number of platforms. Details of the hardware configuration of each platform and the compilers used are presented below.

SUN 4/50 with 16 Mbyte of RAM and Sun's SC1.0 Fortran V1.4. The minimal optimisation level is `-O1`, which is post-pass peephole optimisation of assembly code. Maximal optimisation is `-O4`, which includes induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop invariant optimisation, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, complex expression expansion, optimisation of references and definitions of external variables, and automatic inlining of functions contained in the same file.

DEC 3300L with 96 Mbyte of RAM and version 3.5 of the DEC Fortran on AXP OSF/1 compiler system. Minimal optimisation `-O0` disables all optimisations. Maximal optimisation `-O4` enables local optimisations, recognition of common subexpressions, code motion, strength reduction and test replacement, split lifetime analysis, code scheduling, inlining of arithmetic statement functions, integer multiplication and division expansion (using shifts), loop unrolling, code replication to eliminate branches, and inline expansion of small procedures.

Operation	*Time, μs*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
t_{loopoh}	0.518	0.273	0.230
$t_{\text{load}I}$	0.168	0.244	0.0305
$t_{\text{store}I}$	0.159	0.159	0.0188
$t_{\text{add}I}$	0.0905	0.0325	0.00297
$t_{\text{mult}I}$	4.42	0.156	0.0680
$t_{\text{load}R}$	0.135	0.201	0.0255
$t_{\text{store}R}$	0.235	0.172	0.0326
$t_{\text{add}R}$	0.0108	0.106	0.00744
$t_{\text{mult}R}$	0.00916	0.116	0.0208

Table 2–5: Basic operation times for unoptimised code

SGI Indigo2 with 96 Mbyte of RAM using MIPS FORTRAN 77 release 5.0. Minimal optimisation level is `-O0` which disables all optimisations. Maximal optimisation level `-O2` performs undefined "global optimisation".

2.3 Predicting the Execution Time of Code Fragments

In this section, the prediction methodology introduced above is applied to modified versions of six of the Livermore Loops, over a variety of problem sizes. These predictions are then compared with measured execution times on each of the three platforms characterised above.

The six codes have been slightly modified in two ways: single precision `REAL`s are used in place of the double precision variables in the original versions; and unlike the originals, the fragments are parameterised on vector length, so that the upper loop bound can be varied.

Fragment A (Loop 1 of the Livermore Loops) is a fragment from a hydrodynamics code:

```
      DO I=1,N
        X(I) = Q + (Y(I) * ((R * Z(I+10)) +
     &                      (T * Z(I+11))))
      ENDDO
```

Fragment B (Loop 3 of the Livermore Loops) is the inner product function, which frequently occurs in scientific codes.

```
      Q = 0.0
      DO I=1,N
        Q = Q + (Z(I)*X(I))
      END DO
```

Fragment C (Loop 5 of the Livermore Loops) is a fragment of a tridiagonal elimination routine.

```
      DO I=2,N
        X(I) = Z(I) * (Y(I)-X(I-1))
      END DO
```

Fragment D (Loop 7 of the Livermore Loops) is taken from an Equation of State code.

```
      DO I=1,N
        X(I) = U(I) + (R * (Z(I) + (R * Y(I)))) +
     &         (T * (U(I+3) + (R * (U(I+2) +
     &                             (R * U(I+1))))) +
     &         (T * (U(I+6) + (R * (U(I+5) +
     &                             (R * U(I+4)))))))
      END DO
```

Fragment E (Loop 11 of the Livermore Loops) is a prefix sum. The value of `x(i)` is set to `y(1) + ... + y(i)`.

```
      X(1) = Y(1)
```

Basic Operation	*Coefficient*					
	Loop A	*Loop B*	*Loop C*	*Loop D*	*Loop E*	*Loop F*
loopoh	n	n	$n-1$	n	$n-2$	$2n$
load_I	0	0	0	0	0	0
store_I	0	0	0	0	0	0
add_I	0	0	0	0	0	$2n$
mult_I	0	0	0	0	0	0
load_R	$3n$	$2n$	$3n-3$	$9n$	$2n-1$	$6n$
store_R	n	0	$n-1$	n	$n-1$	$2n$
add_R	$2n$	n	$n-1$	$8n$	$n-1$	$4n$
mult_R	$3n$	n	$n-1$	$8n$	0	$2n$

Table 2–6: Operation counts for the code fragments

```
DO I=2,N
  X(I) = X(I-1) + Y(I)
END DO
```

Fragment F (Loop 19 of the Livermore Loops) is a general linear recurrence equation.

```
DO I=1,N
  B(I) = T(I) + S * U(I)
  S = B(I) - S
END DO

DO I=1,N
  K = N - I + 1
  B(K) = T(K) + S * U(K)
  S = B(K) - S
END DO
```

The operation counts c_{op} for each of the code fragments are presented as a function of problem size in Table 2–6. We predict the execution time of a fragment as

$$\sum_{\text{op}} (c_{\text{op}} \times t_{\text{op}}) \tag{2.2}$$

where the values t_{op} are those presented in Table 2–5 above. Table 2–7 assesses the accuracy of performance predictions for the six code fragments with a variety of vector lengths on three platforms. The results are summarised by loop fragment in Table 2–8 and by platform in Table 2–9.

The prediction differences given are the difference between the predicted and measured execution times for the fragment, as a percentage of the measured value. A positive prediction difference implies an overestimate by the prediction, and a negative result indicates an underestimate. The measured execution time of a code fragment takes account of both clock resolution and the time taken to read the clock. The effective clock resolution is obtained by repeatedly reading the clock until a difference is observed between subsequent readings. When we measure the execution time of a code fragment, we compare the elapsed time with the clock resolution. If the clock is more than 1% of the elapsed time, we repeat the measurement, this time executing the code fragment a number of times, and calculating the average time taken for a single execution of the fragment. No attempt was made to re-initialise the memory hierarchy between successive executions of a code fragment; there may be register and cache re-use in subsequent executions. We begin by measuring the execution time of a code fragment which simply reads the timer. By subtracting this from the elapsed period between two readings of the timer, we can remove one overhead of measurement, and this is done when measuring the execution time of other code fragments. The basic operation times used as parameters in the performance predictions are derived, as discussed above, from the average execution time of single statement code fragments on a variety of vector lengths.

A striking similarity across all loops and platform is the way the prediction changes relative to the measurement as the vector length is increased. The predicted execution rises relative to the measurement at first, then falls as longer vectors are used. This behaviour suggests a combination of two conflicting factors.

Fragment	*Vector Length*	*Prediction Difference (%)*		
		SUN 4/50	*DEC 3300L*	*SGI Indigo2*
A	10	-18	-14	-3
A	100	-16	-11	+2
A	1000	-15	-11	-1
A	10000	-25	-13	-15
A	100000	-24	-26	-25
B	10	-17	-18	+2
B	100	-8	-12	+11
B	1000	-6	-13	+11
B	10000	-20	-15	-10
B	100000	-21	-29	-33
C	10	+19	+28	+17
C	100	+31	+41	+26
C	1000	+33	+43	+21
C	10000	+3	+10	-4
C	100000	+1	+9	-19
D	10	-31	+31	-5
D	100	-30	+33	-3
D	1000	-29	+34	-5
D	10000	-35	+22	-18
D	100000	-35	+22	-31
E	10	+26	+16	+55
E	100	+40	+29	+74
E	1000	+41	+30	+76
E	10000	+14	+5	+26
E	100000	+9	+1	+8
F	10	-1	+9	-17
F	100	+4	+14	-14
F	1000	+4	+14	-16
F	10000	-10	-1	-30
F	100000	-14	-5	-38

Table 2–7: Prediction differences for unoptimised code. Percentages are calculated relative to the measured execution time.

Difference Metric (%)	*Fragment*					
	A	*B*	*C*	*D*	*E*	*F*
Maximum	+2	+11	+43	+34	+76	+14
Minimum	-26	-33	-19	-35	+1	-38
Average Magnitude	15	15	20	24	30	13
Std Dev Magnitude	8	8	31	11	23	10

Table 2–8: Summary of micro-analysis prediction differences by fragment for unoptimised code

Difference Metric (%)	*Platform*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
Maximum	+41	+43	+76
Minimum	-35	-29	-38
Average Magnitude	19	19	21
Std Dev Magnitude	12	11	19

Table 2–9: Summary of micro-analysis prediction differences by platform for unoptimised code

Recall that we may time a number of executions of a code fragment to ensure that the clock resolution does not unduly perturb our results. For smaller vector lengths, more executions will be required. Hence more iterations of an outer loop around the code fragment will be required, and these will account for part of the measured time, raising it artificially. As the vector length is increased, the number of times the outer loop is required will be reduced, and the artificial overhead in the measured time will fall accordingly. As a result, a model which underestimates execution time for the shortest vector length will tend to underestimate by less, or even overestimate, as the vector length is increased, while a model which initially overestimates will overestimate by a larger proportion as the vector length rises towards the middle of the range. The relative fall in measured time with respect to the model can be seen in every data item in the table.

The fall in the predictions relative to the measurements, for larger vector lengths, is due to cache effects. For smaller vector lengths, only the first run of the loop will cause cache misses, and the time these take will be amortized over all the subsequent runs of the loop. Since none of the arrays are powers of two in length, we do not expect cache conflicts between multiple arrays to cause any performance effects. Above a certain vector length, however, the arrays can no longer be held in cache from execution to execution of the code fragment, and cache misses occur in each execution. This leads to increased execution time above this vector length. Since the prediction takes no account of this effect, we see it fall relative to measured execution time for larger vector lengths.

Comparing the accuracy of predictions for the various fragments it is interesting to note that the execution time of Fragment E is generally over-predicted. Table 2–8 shows that this is the worst fragment for our predictions. In fact, the element of X used in one iteration of fragment E can be reused in the next, requiring only a register load rather than a load from memory. This means that the fragment performs only n loads rather than the $2n - 1$ loads which our initial model assumes. Recalculating the prediction on this basis reduces the average error magnitude from 30% to 19%, which is much more in keeping with the accuracy obtained for the other fragments.

Fragment	*flop count*
A	$5n$
B	$2n$
C	$2n$
D	$16n$
E	$n-1$
F	$6n$

Table 2–10: Flop counts for code fragments

There are also differences between platforms for a given fragment. The most striking contrast is between the SUN 4/50 and the DEC 3300L for Fragment D. The fragment's performance is systematically underpredicted for the SUN and overpredicted for the DEC. This fragment has the largest loop body of any of the fragments, and there is considerable potential for superscalar execution if floating point operations can be overlapped. The long pipelines in the DEC system's Alpha processor can exploit this to achieve high performance. Since the basic operation times were derived from small code fragments with very limited scope for superscalar execution, our prediction derived from these operation times will be an over-estimate. The SUN, on the other hand, has much more limited potential for overlapping operations, and architectural constraints limit its ability to overlap floating point operations. At best, the processor can issue two floating point operations every three cycles, and only by alternating add and multiply operations. The small code fragments used to obtain basic operation times do not capture this pipeline stall effect, so predictions based on them tend to under-predict the execution time of larger code fragments.

We assess the accuracy of our performance predictions through comparison with predictions based solely on Mflop/s rates and flop counts. Table 2–10 presents the characterisations of each of the fragments on the basis of flop counts. Table 2–11 summarises the prediction differences obtained from using these flop counts and the geometric mean Mflop/s rate from the Livermore Loops (Table 2–1). Comparing these figures with Table 2–9, we can see that the micro-analysis approach is significantly more accurate.

Difference Metric (%)	*Platform*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
Maximum	+254	+251	+201
Minimum	-30	-30	-53
Average Magnitude	84	73	51
Std Dev Magnitude	77	76	59

Table 2–11: Summary of Mflop prediction differences by platform for unoptimised code

Metric	*SUN 4/50*	*DEC 3300L*	*SGI Indigo2*
Maximum (Mflop/s)	10.16	46.25	43.87
Geometric Mean (Mflop/s)	4.10	12.77	18.85
Minimum (Mflop/s)	1.41	3.54	5.54
Maximum/Geometric Mean	2.48	3.62	2.33
Minimum/Geometric Mean	0.34	0.27	0.29

Table 2–12: Mflop/s rates from Livermore Loops with maximal compiler optimisation

2.4 Compiler Optimisation

Compiler optimisations can generate very significant performance gains. Table 2–12 shows the Mflop/s rates obtained from the Livermore Loops with full compiler optimisation, while Figure 2–2 compares the performance levels achieved with and without compiler optimisation. For every platform, enabling compiler optimisations more than doubles the maximum performance, and in one case increases it by more than an order of magnitude. Clearly, a performance prediction methodology must be applicable to optimised code.

We will now apply our performance prediction method to optimised codes. The basic operation times obtained with compiler optimisation enabled are presented in Table 2–

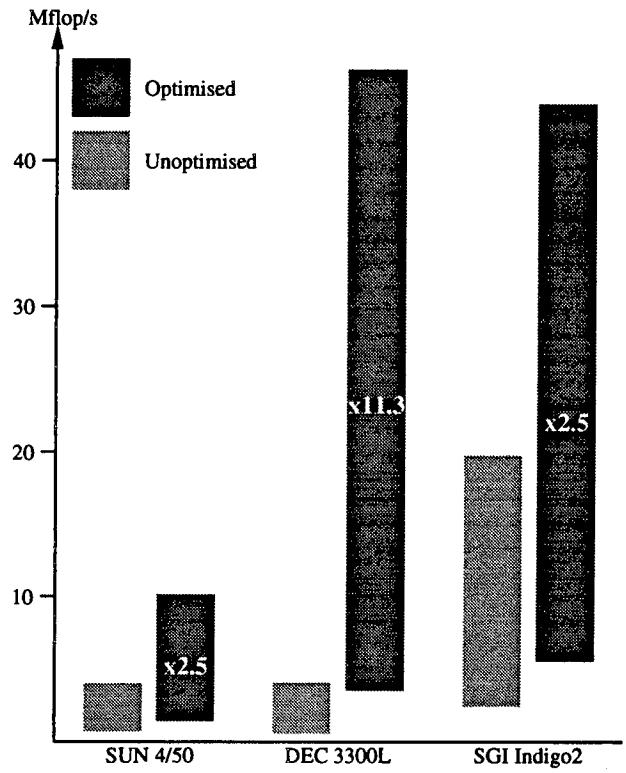


Figure 2–2: Effect of compiler optimisation on floating point performance range, as measured by the Livermore Loops. The superimposed factors show the increase in maximum performance as a result of enabling optimisation.

Operation	*Time, μs*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
t_{loopoh}	0.0972	0.0232	0.0222
$t_{\text{load}I}$	0.0367	0.0250	0.0148
$t_{\text{store}I}$	0.109	-0.00136	0.0112
$t_{\text{add}I}$	0.0566	0.0175	0.0113
$t_{\text{mult}I}$	4.41	0.132	0.0754
$t_{\text{load}R}$	-0.0342	-0.0220	0.00226
$t_{\text{store}R}$	0.179	0.0637	0.0236
$t_{\text{add}R}$	0.185	0.0501	0.0393
$t_{\text{mult}R}$	0.186	0.0504	0.0433

Table 2–13: Basic operation times for optimised code

13. A number of the operation times are negative. This arises because compiler optimisations can reduce the operation counts below the values in Table 2–6 which we use as coefficients in our model fitting procedure. The appearance of negative operation times is generally accepted in this form of analysis and is not simply an artefact of the set of simultaneous equations we have chosen.

The prediction differences resulting from the basic operation times in Table 2–13, expressed as percentages of the optimised codes' execution times, are shown in Table 2–14, and summarised in Tables 2–15 and 2–16.

For fragments B, C and E, the DEC 3300L platform is under-predicted while the SUN and SGI platforms are over-predicted. The DEC 3300L owes its performance to very long pipelines which are inhibited by the loop-carried dependences and small loop bodies in these fragments. Many of the fragments used to derive the basic operation times do not have loop-carried dependences, leading to a under-estimation of execution time for the DEC 3300L. The other platforms are not as susceptible to this effect.

Considering the variation in prediction difference for a given loop as vector length changes, we see the same phenomenon as we encountered in the predictions for un-

Fragment	*Vector Length*	*Prediction Difference (%)*		
		SUN 4/50	*DEC 3300L*	*SGI Indigo2*
A	10	+16	+73	+55
A	100	+24	+203	+108
A	1000	+24	+52	+96
A	10000	+4	+5	+47
A	100000	+2	-36	+28
B	10	+51	-32	+28
B	100	+72	-0	+103
B	1000	+67	-36	+108
B	10000	+7	-51	+18
B	100000	-5	-67	-38
C	10	+9	-25	+17
C	100	+19	-18	+69
C	1000	+21	-40	+60
C	10000	-21	-54	+9
C	100000	-25	-71	-13
D	10	+63	+154	+40
D	100	+65	+270	+44
D	1000	+65	+139	+39
D	10000	+43	+98	+26
D	100000	+43	+42	+8
E	10	+8	-3	+17
E	100	+17	+8	+87
E	1000	+19	-25	+98
E	10000	-21	-52	+15
E	100000	-24	-72	+15
F	10	+40	-12	+39
F	100	+46	+0	+80
F	1000	+45	-19	+74
F	10000	+14	-29	+30
F	100000	+0	-61	+11

Table 2–14: Prediction differences for optimised code

Difference Metric (%)	*Fragment*					
	A	*B*	*C*	*D*	*E*	*F*
Maximum	+203	+108	+69	+270	+98	+80
Minimum	-36	-67	-71	+8	-72	-61
Average Magnitude	51	45	31	76	32	33
Std Dev Magnitude	51	32	21	65	29	24

Table 2–15: Summary of micro-analysis prediction differences by fragment for optimised code

Difference Metric (%)	*Platform*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
Maximum	+72	+270	+108
Minimum	-25	-72	-38
Average Magnitude	29	58	47
Std Dev Magnitude	21	60	32

Table 2–16: Summary of micro-analysis prediction differences by platform for optimised code

Difference Metric (%)	*Platform*		
	SUN 4/50	*DEC 3300L*	*SGI Indigo2*
Maximum	+119	+569	+111
Minimum	-53	-77	-40
Average Magnitude	47	99	42
Std Dev Magnitude	38	135	29

Table 2–17: Summary of Mflop prediction differences by platform for optimised code

optimised code — an initial rise of the prediction relative to the measurement, then a fall. As before, this is due to the interaction between reducing overheads and increasing cache and memory effects as the vector size increases.

Table 2–16 shows that the execution times of the fragments are predicted most accurately for the SUN 4/50. Given this platform's comparatively simple processor architecture, with limited scope for superscalar execution, it is perhaps not surprising that optimisation does not disturb our predictions as much as for the DEC 3300L or the SGI Indigo2.

Table 2–17 shows the accuracy of predictions based on the flop counts, using the geometric mean of optimised Livermore Loops.

Comparison of Tables 2–17 and 2–16 shows that the micro-analysis approach is at least comparably accurate in all cases, and much more accurate in most, than the simpler rate-based prediction.

2.5 Conclusions and Related Work

Although the micro-analysis approach is better than a simple flop-based calculation, prediction differences can still be large. The worst results are obtained for Fragment D with maximal optimisation, for which the average difference magnitude is 76%.

Closer examination of Fragment D (reproduced below) suggests two reasons for the over-estimated execution time, which are weaknesses of this approach.

```
      DO I=1,N
        X(I) = U(I) + (R * (Z(I) + (R * Y(I)))) +
     &          (T * (U(I+3) + (R * (U(I+2) +
     &                               (R * U(I+1))))) +
     &          (T * (U(I+6) + (R * (U(I+5) +
     &                               (R * U(I+4)))))))
      END DO
```

Firstly, we note that there is significant potential for register/cache re-use, which the micro-analysis approach does not capture. After the first six iterations, all the required elements of `U` other than `U(I+6)` can be available in registers, not requiring a load. This reduces the load count per iteration from 9 to 3, with an additional 6 during the first iteration.

Secondly, we note that there is significant recalculation in this loop which an optimising compiler may exploit. We note that:

```
T * (U(I+3) + (R * (U(I+2) + (R * U(I+1)))))
```

need only be calculated for the first three iterations; thereafter the value of:

```
T * (U(I+6) + (R * (U(I+5) + (R * U(I+4)))))
```

from a previous iteration can be reused. This reduces the multiply count from $8n$ to $5n + 9$ where n is the iteration count, and lowers the add count from $8n$ to $6n + 6$.

Recalculating the predictions on this basis reduces the average difference magnitude to 45%, but large differences remain, particularly for the DEC 3300L. The code fragments which are used to derive the basic operation times all involve small efficiently-coded loop nests, which limit a compiler's scope for optimisation, and for super-scalar scheduling. As a result, our basic operation times may not be representative of codes such as Fragment D, which involve larger loop nests offering significant scope for pipelined execution.

Three other issues which our approach does not address are control flow determination, memory hierarchy effects and instruction ordering. We assume that we have perfect knowledge about which statements will be executed and how often they will be executed. In general, this information will depend upon the input data for a particular execution of an application. Some performance prediction work makes arbitrary assumptions in order to determine control flow and loop trip counts [Balasundaram *et al.*, 1990; Kennedy *et al.*, 1991]. Others, notably [Fahringer, 1993; Fahringer, 1994] rely on statistics gleaned from profiling runs. Program slicing techniques [Weiser, 1984], which remove all operations which do not affect control flow, can be used to reduce the execution time of a profiling run. Static analysis techniques have been developed and, in some cases, shown to be competitive with dynamic information from profiling [Wagner *et al.*, 1994]. [de Ronde *et al.*, 1994] describes a Fortran 77 analysis tool which produces symbolic expressions for execution time by propagating constants to characterise loop bounds as constants or simple functions of parameters. Other studies, including [Wall, 1990], evaluate the accuracy of static estimation and profiling approaches for branch prediction.

The memory hierarchy can result in a number of performance effects. Register spills in which, for example, a loop nest requires more temporaries than are available in the processor, can result in a marked performance degradation as values are written to and read from memory. The same sort of discontinuity is often observed when a problem size grows to the point where the data can no longer be held in cache, and the effective load time for a value is increased as cache misses occur. Different cache alignments of data in cache can have marked effects on performance. On a single node of the Cray

T3D, for example, a well cache-aligned implementation of SAXPY can deliver more than three times the performance of an implementation in which cache conflicts occur. Researchers resort to detailed cache simulation [Dunlop and Hey, 1993] or analytical models of limited applicability [Fahringer, 1993] to obtain performance estimates which take account of these effects.

Finally, our approach predicts equal performance for two programs involving the same numbers of operations, irrespective of dependencies between them. Such dependencies can inhibit the super-scalar execution on which many modern microprocessors rely for high performance, potentially resulting in very different performance for two different orderings of the same set of operations. A bin-packing approach [Wang, 1994] based on earlier micro-analysis work [Wang, 1990] yields good results on IBM RS/6000 processors.

In the next chapter we assess the accuracy with which we can predict the second component of a parallel application's execution time — interprocess communication.

Chapter 3

Predicting the Execution Time of Parallel Application Structures

In Chapter 2 we used micro-analysis techniques to predict the execution time of sequential code fragments. We now move on to consider how micro-analysis techniques can be applied to parallel programs. We can extend our micro-analysis techniques treatment to those Single Program Multiple Data (SPMD) programs in which each process is mapped onto a separate processor and all communications are collective — i.e. all processes are involved. In Section 3.1 we review the methods used by programmers and parallelisation tools to develop SPMD parallel applications, and discuss a number of common parallel application structures which can be implemented using collective communications. We measure the execution time of these collective communication operations on a number of parallel platforms in Section 3.2, and characterise their performance numerically and symbolically. In Section 3.3 we use these characterisations to predict the execution time of parallel applications. Finally, in Section 3.4, we assess the accuracy of the predictions and the effectiveness of this approach.

3.1 Parallel Applications

The process of developing a parallel implementation of an application has three stages:

1. Identifying the available parallelism;
2. Structuring the available parallelism;
3. Mapping the structured parallelism onto processors

In the following sections we consider each of these stages in turn. Our starting point will be a sequential computation, which we will represent as a graph.

Definition 3.1.1 *A L-labelled directed acyclic graph is a tuple* (V, E, l_V) *where*

- V *is the set of vertices in the graph,*
- $E \subseteq \mathbb{F}\ (V \times V)$ *is a set of ordered pairs of vertices representing the directed edges in the graph, which satisfies*

$$\forall v \in V. \neg \exists \{v_1, \ldots, v_n\} \subseteq V. \{(v, v_1), (v_1, v_2), \ldots, (v_n, v)\} \subseteq E$$

- $l_V : V \rightarrow L$ *maps the vertices of the graph on the label set* L.

Assuming a set of imperative operations $\mathcal{O}$, we can model a sequential computation as an $\mathcal{O}^+$-labelled DAG $(V, <_{\text{seq}}, l)$ where V is the set of components of the computation, $<_{\text{seq}}$ corresponds to the sequential execution order of these components, and $l : V \rightarrow \mathcal{O}^+$ gives the operation or series of operations associated with each component. The approach we discuss below is applicable to components at a range of granularities, from individual machine instructions to substantial code modules, or even application instances.

As in Chapter 2, we assume perfect information about control flow, so our sequential computation is a linear sequence of operations. Hence, $\forall v \in V$

1. $|\{v'|(v, v') \in <_{\text{seq}}\}| \leq 1$, and
2. $|\{v'|(v', v) \in <_{\text{seq}}\}| \leq 1$, and
3. there is a path through all $v \in V$

Given a cost function $T : \mathcal{O}^+ \rightarrow \mathcal{T}$ which maps operation sequences to a time domain $\mathcal{T}$, we can calculate the cost of the sequential program (V, E, l_V) as:

$$\sum_{v \in V} T(l_V(v)) \tag{3.1}$$

In the next section, we consider how parallelism can be detected in this representation of a computation.

3.1.1 Identifying Available Parallelism

Parallelism is constrained by dependencies between operations which force certain operations to be performed in a certain order. We assume that our imperative operations $\mathcal{O}$ manipulate memory locations $\mathcal{M}$, and that dependencies between operations arise from these memory manipulations.

Definition 3.1.2 *The function* Read : $\mathcal{O}^+ \rightarrow \mathbb{F}(\mathcal{M})$ *gives the set of memory locations which are read by an operation sequence.*

Definition 3.1.3 *The function* Write : $\mathcal{O}^+ \rightarrow \mathbb{F}(\mathcal{M})$ *gives the set of memory locations which are written by an operation sequence.*

Definition 3.1.4 $\ll_{\text{seq}}$ *is the transitive closure of* $<_{\text{seq}}$.

Definition 3.1.5 *The set of intermediate nodes* $v_1 \uparrow_{(V,E)} v_2$ *is given by:*

$$\{v | v \in V \wedge v_1 \ll_{\text{seq}} v \wedge v \ll_{\text{seq}} v_2\}$$

Definition 3.1.6 *We say that there is a true dependency (or read-after-write dependency) between two vertices* v_1 *and* v_2 *of an* $\mathcal{O}^+$*-labelled directed acyclic graph* (V, E, l_V)*, and write* $v_1 \delta_{\text{raw}} v_2$*, iff:*

1. $v_1 \ll_{\text{seq}} v_2$,
2. $|(\text{Write}(l_V(v_1)) \cap \text{Read}(l_V(v_2))) - \bigcup_{v \in v_1 \uparrow_{(V,E)} v_2} \text{Write}(l_V(v))| > 0$

Definition 3.1.7 *We say that there is an output dependency (or write-after-write dependency) between two vertices* v_1 *and* v_2 *of an* $\mathcal{O}^+$*-labelled directed acyclic graph* (V, E, l_V)*, and write* $v_1 \delta_{\text{waw}} v_2$*, iff:*

1. $v_1 \ll_{\text{seq}} v_2$,
2. $|(\text{Write}(l_V(v_1)) \cap \text{Write}(l_V(v_2))) - \bigcup_{v \in v_1 \uparrow_{(V,E)} v_2} \text{Write}(l_V(v))| > 0$

Where there is a dependency between two vertices v_1 and v_2 but it is not necessary to distinguish between true and output dependencies we will write $v_1 \delta v_2$.

Definition 3.1.8 *The task graph corresponding to a sequential computation represented by the* $\mathcal{O}$*-labelled directed acyclic graph* (V, E, l_V) *is a tuple*

$$(V, E', l_V, l_\delta)$$

where

$$E' = \{(v_1, v_2) | v_1, v_2 \in V \wedge v_1 \delta v_2\}$$

and $l_\delta : \mathbb{F}\ (V \times V) \rightarrow \mathcal{M}^*$ *represents the memory locations whose values must be communicated between the tasks.*

$$l_\delta(v_1, v_2) = \text{Write}(l_V(v_1)) \cup \text{Read}(l_V(v_2))$$

We can use our cost function T to determine the most expensive path through the task graph. If the cost of this path is determined to be c, then the sequential fraction of the computation is given by

$$\alpha = \frac{c}{(\sum_{v \in V} T(l_V(v)))} \tag{3.2}$$

and the achievable speedup is bounded by $\frac{1}{\alpha}$.

3.1.2 Structuring Available Parallelism

Direct implementation of the task graph as a set of communicating processes is not an attractive option for a number of reasons. Firstly, the overheads of process management are likely to be large relative to the execution time of the computations in each task. This issue is addressed by combining tasks wherever possible, and by mapping a number of tasks into a process (Section 3.1.3). Secondly, the task graph often contains many small communications each involving a small amount of data. For small communications, the overhead of sending a message is large with respect to the time the message spends in transit, and on many architectures this overhead will be significant with respect to the execution time of a task. To address this problem, we will seek to combine messages wherever possible. Finally, some parallel architectures provide hardware support for certain communication patterns, which allow these communications to be realised more efficiently if they are specified as high level operations. The task graph representation uses point-to-point communications and we will seek to transform the structure of the graph to facilitate the recognition of collective communication operations.

There are many transformations which can be applied to task graphs, to increase the available parallelism, or improve the potential efficiency of a parallel implementation [Zima and Chapman, 1990]. For example, a scalar temporary variable can be replaced

with a vector, thereby removing dependencies between tasks. We will restrict our attention to four transformations, but first we introduce the notion of a task graph segment.

Definition 3.1.9 *Δ is defined as the transitive closure of δ.*

Definition 3.1.10 *Two components c_1 and c_2 of a computation are said to be independent if neither $c_1 \Delta c_2$ nor $c_2 \Delta c_1$.*

Definition 3.1.11 *Two operations o_1 and o_2 are said to be similar, written $o_1 \equiv o_2$ if they differ only in their parameters. For example, the operations:*

```
X(1) = Y(1) * Z
```

and

```
X(4) = Y(4) * Z
```

could be regarded as similar. Since much potential parallelism comes from loops, a stricter definition would require two similar operations to be instances of the same statement in a particular loop nest. The appearance of functions, as in

```
X(1) = F(1)
```

and

```
X(4) = F(4)
```

introduces the possibility that $T(o_1) \neq T(o_2)$.

Definition 3.1.12 *A segment S of a task graph (V, E, l_V, l_E) is a set*

$$\{v_1, \ldots, v_n\} \subseteq V$$

such that, $\forall v_i, v_j \in S$

1. $l_V(v_i) \equiv l_V(v_j)$

2. *neither* $v_i \Delta v_j$ *nor* $v_j \Delta v_i$

3. $\forall v \in V$ *such that* $v \notin S$ *then* $v \Delta v_i$ *or* $v_i \Delta v$ *for some* $i \in \{1, \ldots, n\}$

Definition 3.1.13 *A segmentation of a task graph* (V, E, l_V, l_E) *is a set of segments* $\{S_1, \ldots, S_n\}$ *such that*

1. $V = \bigcup_{i=1 \ldots n} S_i$

2. $\forall S_i, S_j \in \{S_1, \ldots, S_n\}. S_i \cap S_j = \emptyset$

Transformation 1: Combining Communications To The Same Segment

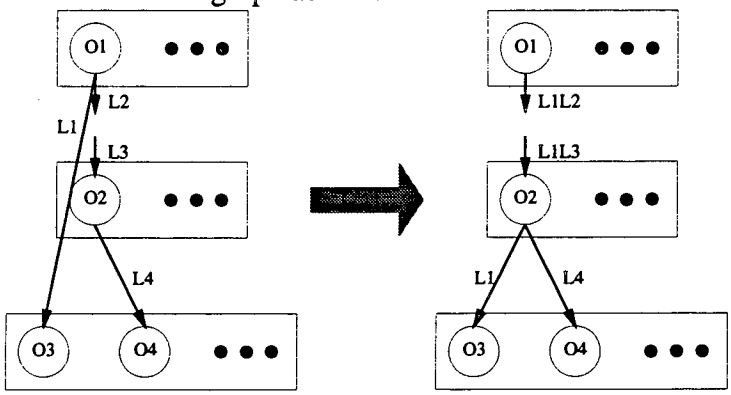
If a task graph (V, E, l_V, l_E) with vertices labelled by $\{O_1, \ldots, O_n\}$ and edges labelled by $\{L_1, \ldots, L_N\}$ contains a pair of edges $(v_1, v_3), (v_2, v_4) \in E$ such that

1. $v_1 \Delta v_2$, and

2. $v_3, v_4 \in S$ where S is a segment of (V, E, l_V, l_E), and

3. there is no $v \in V$ such that $v_1 \Delta v \Delta v_2$ and $|\text{Write}(l_V(v)) \cap l_E((v_1, v_3))| > 0$

then we can transform the task graph as follows:



1. Let $l = l_E((v_1, v_3))$

2. Remove the edge (v_1, v_3)

3. If there is no edge (v_2, v_3) then add it

4. Add l to the label of edge (v_2, v_3)

5. Add l to the labels of all edges on all paths from v_1 to v_2

Transformation 2: Combining Communications From The Same Segment

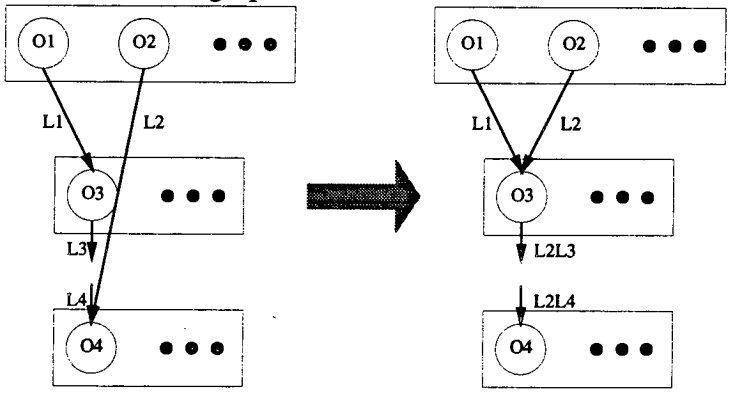
If a task graph (V, E, l_V, l_E) with vertices labelled by $\{O_1, \ldots, O_n\}$ and edges labelled by $\{L_1, \ldots, L_N\}$ contains a pair of edges $(v_1, v_3), (v_2, v_4) \in E$ such that

1. $v_1, v_2 \in S$, where S is a segment of (V, E, l_V, l_E), and

2. $v_3 \Delta v_4$, and

3. there is no $v \in V$ such that $v_3 \Delta v \Delta v_4$ and $|\text{Write}(l_V(v)) \cap l_E((v_2, v_4))| > 0$

then we can transform the task graph as follows:



1. Let $l = l_E((v_2, v_4))$

2. Remove the edge (v_2, v_4)

3. If there is no edge (v_2, v_3) then add it

4. Add l to the label of edge (v_2, v_3)

5. Add l to the labels of all edges on all paths from v_3 to v_4

Transformation 3: Combining Communications in General

If a task graph (V, E, l_V, l_E) with vertices labelled by $\{O_1, \ldots, O_n\}$ and edges labelled by $\{L_1, \ldots, L_N\}$ contains a pair of edges $(v_1, v_3), (v_2, v_4) \in E$ such that

1. $v_1 \Delta v_2$

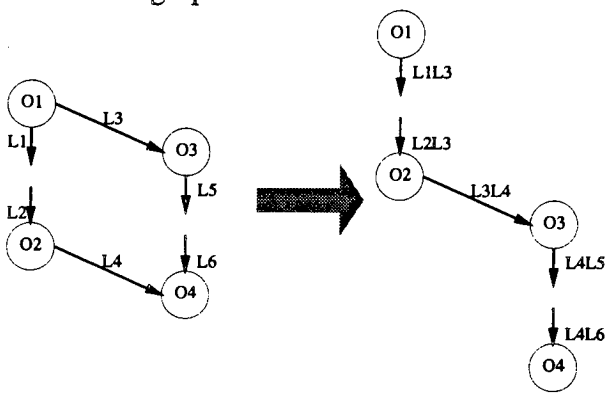
2. $v_3 \Delta v_4$

3. it is not the case that $v_3 \Delta v_2$

4. there is no $v \in V$ such that $v_1 \Delta v \Delta v_2$ and $|\text{Write}(l_V(v)) \cap l_E((v_1, v_3))| > 0$

5. there is no $v \in V$ such that $v_3 \Delta v \Delta v_4$ and $|\text{Write}(l_V(v)) \cap l_E((v_2, v_4))| > 0$

then we can transform the task graph as follows:



1. Let $l_3 = l_E((v_1, v_3))$

2. Let $l_4 = l_E((v_2, v_4))$

3. Remove the edges (v_1, v_3) and (v_2, v_4)

4. Add the edge (v_2, v_3) if it is not present

5. Add $l_3 \cup l_4$ to the label of edge (v_2, v_3)

6. Add l_3 to the labels of all edges on all paths from v_1 to v_2

7. Add l_4 to the labels of all edges on all paths from v_3 to v_4

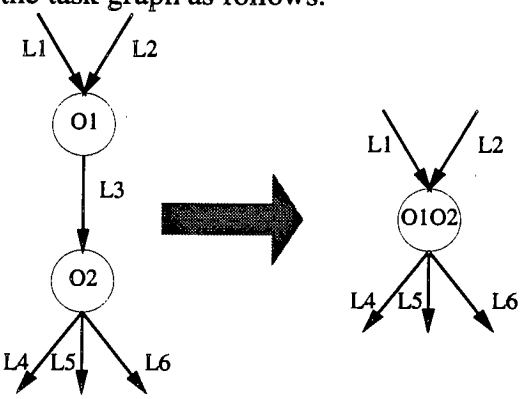
Transformation 4: Combining Tasks

If a task graph (V, E, l_V, l_E) with vertices labelled by $\{O_1, \ldots, O_n\}$ and edges labelled by $\{L_1, \ldots, L_N\}$ contains two vertices $v_1, v_2 \in V$ such that

1. $\{v|(v_1, v) \in V\} = \{v_2\}$

2. $\{v|(v, v_2) \in V\} = \{v_1\}$

then we can transform the task graph as follows:



1. Append $l_V(v_2)$ to $l_V(v_1)$

2. For every edge $(v_2, v) \in V$

   (a) If there is no edge (v_1, v) then add (v_1, v)

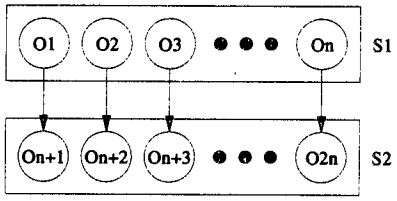
   (b) Add $l_E((v_2, v))$ to the label of (v_1, v)

   (c) Remove (v_2, v)

3.1.3 Mapping Structured Parallelism

Given a task graph, judiciously manipulated by the transformations in Section 3.1.2, we need to map the tasks into processes, one for each processor. We do this by taking a segmentation of the task graph, and considering each segment in turn. There are two factors which must be taken into account when deciding how to map the tasks in a segment: the mapping of other tasks on which tasks in this segment have dependencies; and the load balancing of tasks within the segment.

The mapping of other tasks on which the segment in question has dependencies is important because the relative mapping of the segments determines which, if any, dependencies must be mapped onto communication. For example, the segments $S1$ and $S2$ shown below can be mapped in many ways. In the worst case, some n communications will be necessary, while in the best case no communication is required.



The load balance across tasks in the segment must be taken into account to distribute the work evenly across processes. If, for segment S

$$\forall v_i, v_j \in S.T(l_V(v_i)) = T(l_V(v_j)) \tag{3.3}$$

then we can achieve an even distribution of work by placing as close as possible to the same number of tasks on each processor. If Equation 3.3 does not hold, but $T(l_V(v))$ is known $\forall v \in S$, then a bin-packing approach can be used to achieve as good a load balance as possible. If there is no information available about load balance, then distributing an even number of tasks to each process is the best approach. This relies on statistical averaging to provide good load balance, and works best if the number of tasks is very much larger than the number of processes. Note, however, that some distributions of tasks may achieve much better load balance than others.

Clearly, the objectives of minimising communication and optimising load balance can conflict. Suppose we have a segment $S = \{v_1, \ldots, v_8\}$ with $T(l_V(v_i)) = i$ which we wish to map into four processes. The worst case mapping which places equal numbers of tasks in each process is

$$\begin{array}{llll} m(v_1) = 1 & m(v_2) = 1 & m(v_3) = 2 & m(v_4) = 2 \\ m(v_5) = 3 & m(v_6) = 3 & m(v_7) = 4 & m(v_8) = 4 \end{array} \tag{3.4}$$

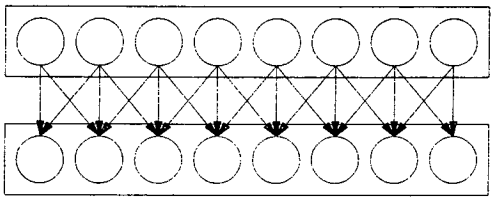
which distributes work into four chunks $\{3, 7, 11, 15\}$. It is possible to achieve a perfect load balance of $\{9, 9, 9, 9\}$ using

$$\begin{array}{llll} m(v_1) = 1 & m(v_2) = 2 & m(v_3) = 3 & m(v_4) = 4 \\ m(v_5) = 4 & m(v_6) = 3 & m(v_7) = 2 & m(v_8) = 1 \end{array} \tag{3.5}$$

Suppose that we have an identical segment $S' = \{v'_1, \ldots, v'_n\}$ of this form, with dependencies

$$\begin{array}{ll} & \bigcup_{i=1\ldots8}\{(v_i, v'_i)\} \\ \cup & \bigcup_{i=2\ldots8}\{(v_i, v'_{i-1})\} \\ \cup & \bigcup_{i=1\ldots7}\{(v_i, v'_{i+1})\} \end{array}$$

as shown below.

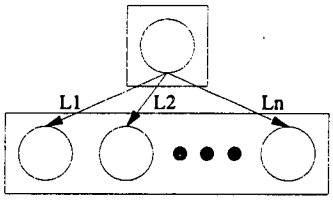


The best mapping from a load balancing point of view (Equation 3.5) gives rise to 12 communications, while the poor mapping (Equation 3.4) requires only 6 . Different mapping algorithms will trade off conflicts of this type in different ways.

Once the task graph has been mapped, communication operations must be inserted. We can ignore all edges between tasks which are mapped onto the same process. As a result of the transformations in Section 3.1.2, a particular value may be carried into a

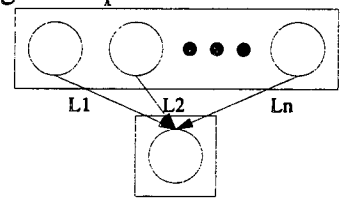
task by several edges. For efficiency, each value should appear on at most one edge, and the value should be removed from the labels of the other edges. In order to avoid unnecessary communication, this removal should be propagated up through preceding nodes in the graph.

We can identify a number of collective communication operations as structures in the graph. If we have segments $S_1 = \{v_0\}$ and $S_2 = \{v_1, \ldots, v_n\}$ with $\{(v_0, v) | v \in S_2\} \subseteq V$, as shown below, then we have a multi-cast or a scatter operation.

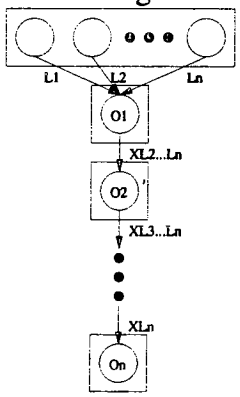


Writing $l_E(v_0, v_i) = l_i$, if $\forall i, j \in \{1 \ldots n\}.l_i = l_j$ then we have a multi-cast. If $\forall i, j \in \{1 \ldots n\}.l_i \neq l_j$, we have a scatter operation. Otherwise, we have a series of at least one scatter and one multi-cast operation.

If we have segments $S_1 = \{v_1, \ldots, v_n\}$ and $S_2 = \{v_0\}$ with $\{(v, v_0) | v \in S_1\} \subseteq V$, as shown below, then we have a gather operation.



Reduction operators can be recognised through the following structure

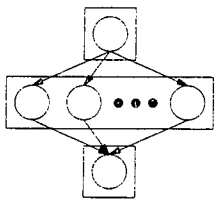


in which $O1 = O2 = \ldots = On = O$ where O is a reduction operation.

3.1.4 Parallel Programming Idioms

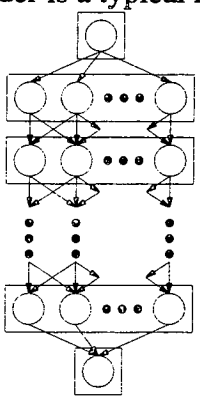
Rather than exploring general mapping techniques, we will concentrate on recognising common task graph structures which are amenable to particular parallel application structures. In the discussion of mapping above, we assumed a static mapping of tasks to processes. Dynamic mapping or re-mapping techniques can also be used, and we will consider them alongside static techniques in this section.

A graph of the form shown below can be mapped statically as a scatter-gather computation, or dynamically as a task farm. This form of graph occurs in many graphics computations, statistical simulations and electro-magnetics models.



In the static case, load balancing the tasks in the parallel segment is achieved as described in the discussion of mapping in Section 3.1.3. In a task farm implementation, a group of tasks are assigned to processes in turn, with each process requesting a further group of tasks when the first group is completed. This continues until there are no more tasks to be executed. A task farm trades off increased communication costs against savings from achieving a better load balance.

The second form of graph we consider is a typical iterative computation:



This structure occurs regularly in partial differential equation solvers used to model phenomena in a wide range of disciplines including fluid dynamics, epidemiology, com-

bustion and thermodynamics. Applications of this form can also be found in image processing. The number of tasks in a subsequent iteration which depend on a particular task is arbitrary. Here, three dependencies are carried forward. Higher numbers of dependencies are possible, with tasks interacting with a larger number of other tasks.

If the parallel segments each include n tasks $\{v_1, \ldots, v_n\}$, and

$$\forall i, j \in \{1 \ldots n\}. T(l_V(v_i)) = T(l_V(v_j))$$

then a regular domain decomposition is used, placing an equal number of tasks on each processor. This is done in such a way as to minimise communication by mapping task sequences which interact into the same process, as far as possible. If the tasks do not involve exactly the same amount of computation, a bin-packing approach can be used to produce an irregular domain decomposition which statically optimises the load balance, provided that the amount of work associated with each task can be predicted. The wider the variation in work associated with a task, the harder it is to achieve a good load balance. If the variance is spatially correlated, so that a particular group of interacting task sequences involve a much higher amount of work than others, it can be important to scatter the task sequences more widely across the processors. Scattered domain decomposition techniques take this approach, seeking to achieve better load balance at the cost of increased communications. If the distribution of work to processes varies during the computation, dynamic re-mapping techniques can be used. If the variation is predictable, a mapping algorithm may identify phases in which different distributions are most appropriate and insert communication operations to achieve the re-mapping. If the variation is not predictable, dynamic load-balancing techniques can be used to re-allocate tasks to processors based on information about the time taken to execute previous iterations on particular processors. Like the other approaches, this method incurs increased communication costs in an effort to reduce the time that is lost due to poor load balance.

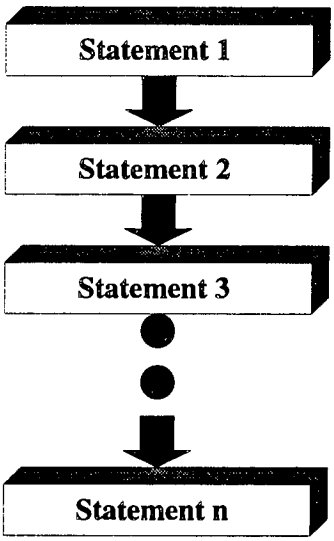


Figure 3–1: Micro-analysis of sequential programs

3.2 Modelling Collective Communications

In our treatment of sequential programs in Chapter 2 we used a linear model constructed from costs associated with each program statement (Figure 3–1). We can extend our micro-analysis techniques to SPMD programs (Figure 3–2).

We can obtain cost models for sequential operations or sequences thereof using the techniques presented in Chapter 2, or with more detailed simulation or empirical methods. We must construct cost models for the collective communication operations:

Throughout this chapter, we will evaluate performance on three parallel platforms:

- a Meiko Computing Surface consisting of T800 transputers with a custom configurable interconnect;
- a collection of SUN IPX workstations connected by Ethernet;
- a Cray Research T3D consisting of Alpha 21064 processors connected by a custom three-dimensional torus network.

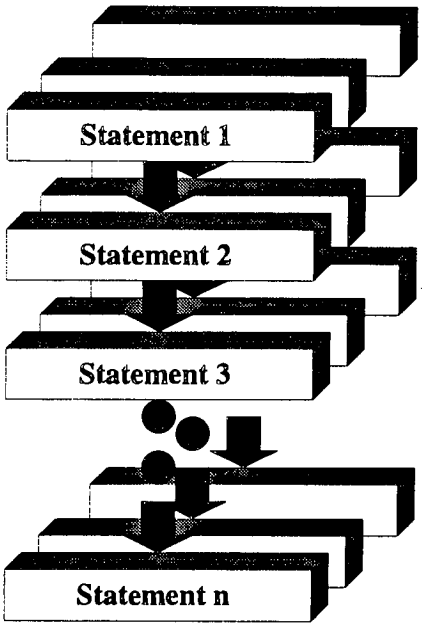


Figure 3–2: Micro-analysis of SPMD parallel programs

We will use MPI as a programming interface on all these platforms.

All collective operations take the form shown in Figure 3–3, in which the collective operation OP may involve synchronisation of processes, motion of data, and computation.

The performance of collective operations typically varies with the number of processors and the amount of data involved in the operation. Given measurements of an operation's performance across a range of processor counts and message sizes, we can predict the performance of an application in two ways. Firstly, we can take a numerical approach, interpolating between (or extrapolating from) measurements to find the execution time of the operation at another point in parameter space. We obtain an estimate of the application's execution time by summing these numerical predictions for the basic operations. This approach promises reasonable accuracy, but it is a cumbersome method for evaluating an application's performance across its parameter space, since each point must be evaluated numerically. Our second approach seeks to address this failing by fitting symbolic models to basic operation performance. We can then derive symbolic expressions for the execution time of an application by summing those of its basic operations.

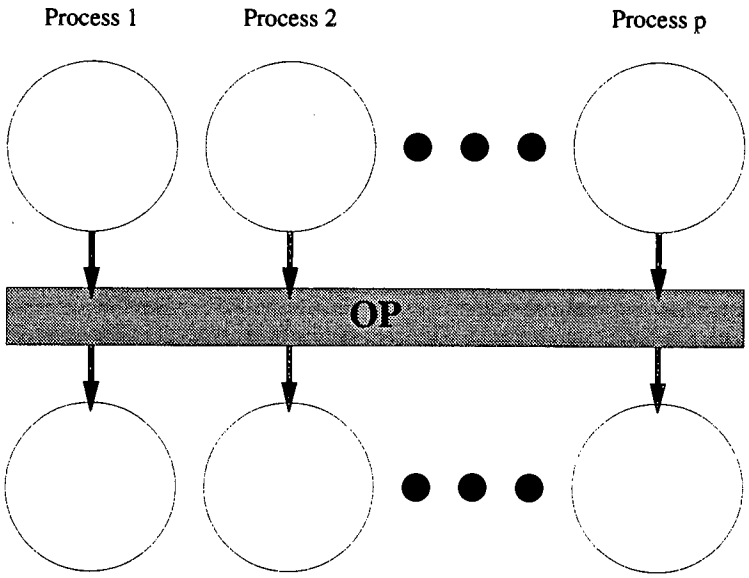


Figure 3–3: Collective communication operation

3.2.1 Barrier

A barrier operation (Figure 3–4) performs a global synchronisation. No process can exit the barrier operation until all processes have entered it. A barrier can be implemented by a central server, with execution time linear in the number of processors, or more efficiently by a process tree, which has execution time logarithmic in the number of processors. A number of parallel systems have hardware support for barriers. When this can be exploited, i.e. when the barrier is not being applied to a subset of the processes, it offers performance which is close to constant time.

The execution times of barrier operations on the Cray T3D, the Meiko Computing Surface and networked workstations are shown in Tables 3–1, 3–2 and 3–3 respectively. This barrier is executed by the statement:

```
CALL MPI_BARRIER(MPI_COMM_WORLD, IERROR)
```

The T3D performance is very close to constant. Modelling the execution time of a barrier as the average of the measured times gives Equation 3.6, which results in an average prediction difference of 4%, with a standard deviation of 2%. Clock resolution on the T3D is approximately 0.006μs.

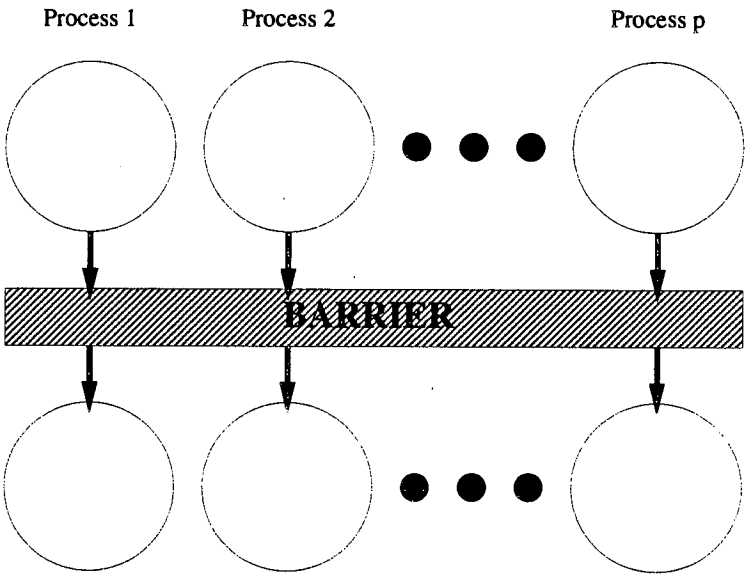


Figure 3–4: Barrier operation

Processors	*Time (μs)*
2	7.857
4	7.166
8	7.166
16	7.177
32	7.187
64	7.197
128	7.970
256	7.188

Table 3–1: Barrier execution time (μs) on the Cray T3D

Processors	*Average Time (μs)*
2	17954
4	22676
8	30165
12	34836
16	36413
20	41523
24	42692
28	42028
32	48415
36	51614
40	51582
44	52174
48	54191
52	50647
56	50647

Table 3–2: Barrier execution time (μs) on the Meiko Computing Surface

Processors	*Average Time (μs)*
2	8219
3	12330
4	14614
5	17312
6	20261
7	20995
8	26690

Table 3–3: Barrier execution time (μs) on networked workstations

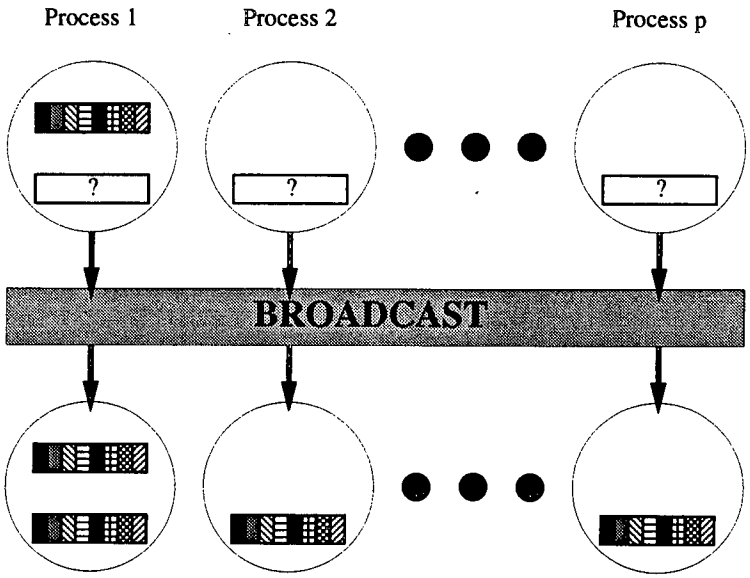


Figure 3–5: Broadcast

$$t_{\text{barrier}}(p) = 7.3635\ \mu s \tag{3.6}$$

On the Meiko Computing Surface, execution time is logarithmic in the number of processors. Equation 3.7 achieves an average prediction difference of 5% with a standard deviation of 3%.

$$t_{\text{barrier}}(p) = 7710 + 7834 \log_2 p\ \mu s \tag{3.7}$$

On the workstation network, execution time is linear in the number of processors. Equation 3.8 results in an average prediction difference of 4% and a standard deviation of 3%.

$$t_{\text{barrier}}(p) = 2305 + 2800p\ \mu s \tag{3.8}$$

3.2.2 Broadcast

In a broadcast operation (Figure 3–5), all processes receive a copy of the data broadcast by a single process. In MPI, broadcast is provided by the `MPI_BCAST` function. MPI's communicator constructs allow the programmer to select a subset of the processes to participate in collective operations. This allows multi-cast to be provided using a broadcast operation.

Although bus-based systems can theoretically broadcast a message in constant time with respect to the number of processors in the system, the transport protocols used typically preclude this. The transport protocols typically support only point-to-point communication, forcing broadcast to be implemented by a number of point-to-point operations, each of which incurs the startup overhead of the whole protocol stack. As a result, a broadcast message may well be sent a number of times. Broadcast can be implemented naively as repeated sends, with an execution time linear in the number of processors, or as a broadcast tree (logarithmic in the number of processors).

Here we consider the time taken to execute:

```
      CALL MPI_BCAST(BUFFER, SIZE, MPI_INTEGER, 0,
     &               MPI_COMM_WORLD, IERROR)
```

for a variety of values of `SIZE` on various numbers of processors. The execution times of broadcast operations on the Cray T3D, the Meiko Computing Surface and networked workstations are shown in Tables 3–4, 3–5 and 3–6 respectively.

The T3D performance is logarithmic in the number of processors. Equation 3.9 gives an average prediction difference of 21% with a standard deviation of 19%.

$$t_{\text{broadcast}}(p,d) = -86.0016 + 232.4384\log_2 p - 0.33357d + 0.243861d\log_2 p\ \mu s \quad (3.9)$$

The Meiko Computing Surface's performance is also logarithmic in the number of pro-

Processors	*Message size (integers)*						
	1	*10*	*100*	*1000*	*10000*	*100000*	*1000000*
2	164.0	165.5	173.2	253.4	819.3	6463.7	62772.5
4	327.8	330.2	345.2	504.5	1969.0	16599.1	162535.3
8	558.8	567.0	596.7	952.2	3381.3	33133.9	325002.8
16	833.3	844.2	888.2	1476.6	6547.6	57307.9	563334.3
32	1109.3	1121.0	1183.6	1946.2	8274.8	72928.6	718823.4
64	1383.8	1398.3	1475.6	2479.7	12779.1	115455.5	1142868.9
128	1658.6	1675.1	1771.6	3024.3	15520.9	143530.1	1426177.5
256	1933.8	1955.5	2062.4	3562.8	17985.1	173613.6	1716588.5

Table 3–4: Broadcast execution time (μs) on the Cray T3D

Processors	*Message size (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	9228	9243	9453	12759	41252	324650
4	13164	13171	13554	23790	132728	1209736
8	16076	16169	16765	34107	222863	2128982
12	16286	16475	17836	35502	244901	2116355
16	17928	17978	18883	40723	261920	2491865
20	19844	19839	21100	41141	284820	2755157
24	15081	14972	16971	41398	209131	2973531
28	23338	23374	23960	45875	314437	2972185
32	16074	16143	17216	57742	511050	5124987
36	18238	18591	20750	53032	392707	3909447
40	19586	19314	21348	56014	446626	4399467
44	14271	14462	16574	51745	432525	4222030
48	13124	13242	15208	48562	379612	3679321
52	23001	23066	24949	69993	568772	5560321
56	23140	23281	25532	68379	555780	5393224

Table 3–5: Broadcast execution time (μs) on Meiko Computing Surface

Processors	*Message size (integers)*						
	1	*10*	*100*	*1000*	*10000*	*100000*	*1000000*
2	4944	5278	5504	14413	113755	1092331	11370598
3	5982	6095	6739	21759	215610	2154516	21402239
4	7829	7908	8343	27774	265653	2569173	26256060
5	9420	11308	11950	33308	388148	3672572	37916688
6	8659	9102	10262	38306	319139	3136036	31410921
7	10537	10729	11950	43312	412703	4099542	40976694
8	10871	10960	12254	63805	484000	4338764	42978941

Table 3–6: Broadcast execution time (μs) on networked workstations

cessors. Equation 3.10 achieves an average prediction difference of 18% with a standard deviation of 15%.

$$t_{\text{broadcast}}(p,d) = 10403.51 + 795.3233 \log_2 p - 10.0613d + 0.9814888d \log_2 p \ \mu s \quad (3.10)$$

Very large variations were observed in the execution time of a given broadcast operation on the workstation network. The best fit to the measurements is achieved with a logarithmic model, but Equation 3.11 results in an average prediction difference of 38%.

$$t_{\text{barrier}}(p) = -15954 - 3.56395d + 8642.042 \log_2 p + 15.50546d \log_2 p \ \mu s \quad (3.11)$$

3.2.3 Scatter

In a scatter operation (Figure 3–6), separate data parcels from one scattering processor are distributed to each of the processors. In MPI, this functionality is provided by the `MPI_SCATTER` operation. Scatter can be implemented by the scattering processor sending a message to each of the other processors containing their fragment of the data.

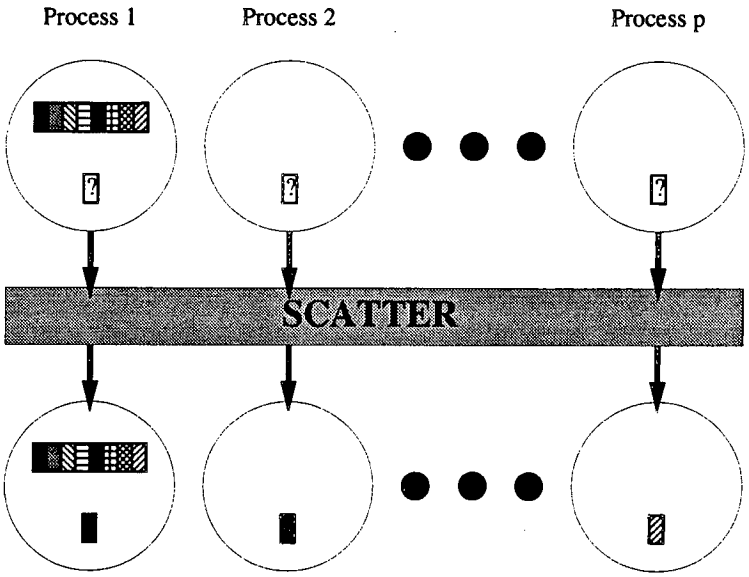


Figure 3–6: Scatter

An alternative implementation of a scatter involves distributing the data down through a logical b-ary tree of processors.

Here we consider the time taken to execute:

```
      CALL MPI_SCATTER(OUTBUF, SIZE*P, MPI_INTEGER,
     &                 INBUF,  SIZE,   MPI_INTEGER,
     &                 0, MPI_COMM_WORLD, IERROR)
```

for a variety of values of `SIZE` on various numbers of processors `P`. The execution times of scatter operations on the Cray T3D, the Meiko Computing Surface and networked workstations are shown in Tables 3–7, 3–8 and 3–9 respectively. The memory required by a scatter of i integers to p processors is dominated by the program buffer space ip. For larger numbers of processors and larger messages, this can exceed the memory available on a single node of the system. This is the case with the Cray T3D and the Meiko Computing Surface, so Tables 3–7 and 3–8 are necessarily incomplete.

The execution time of scatter on the T3D is linear in the number of processors. Equation 3.12 gives an average prediction difference of 5% and a standard deviation of 3%.

Processors	*Message size per processor (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	323.4	325.0	326.5	530.8	1987.4	16737.4
4	550.5	556.0	591.0	937.8	3525.5	29560.0
8	1000.7	1013.0	1082.1	1751.3	6597.8	55203.0
16	1903.2	1928.3	2064.0	3374.8	12740.7	
32	3705.8	3749.3	4022.4	6626.4	25025.6	
64	7283.3	7376.1	7918.2	13179.8	50263.2	
128	14494.3	14674.4	15754.1	26258.8		
256	28954.9	29330.7	31495.2	52405.0		

Table 3–7: Scatter execution time (μs) on Cray T3D

Processors	*Message size per processor (integers)*				
	1	*10*	*100*	*1000*	*10000*
2	12591	12605	12632	15796	47926
4	16954	16971	17348	28308	141547
8	23813	23865	25078	51230	310359
12	37199	37231	43844	82706	
16	44659	44816	47973	103800	
20	59701	59298	67911	130923	
24	66495	66616	79976	167317	
28	80426	80163	92421	191513	
32	78918	78898	85174	206434	
36	100930	101016	116315	258747	
40	116144	116533	134538	288093	
44	121212	121354	142236	315398	
48	133854	134721	152409	337642	
52	146567	147891	172495	397791	
56	160316	162940	187171	439199	

Table 3–8: Scatter execution time (μs) on the Meiko Computing Surface

Processors	*Message size per processor (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	6469	5862	6319	15070	118839	1181185
3	7893	7375	7943	19185	222253	2193559
4	9665	9108	9821	32246	271474	2639614
5	12443	11994	12798	48049	436280	4341517
6	13235	12889	14440	52165	495128	4926294
7	16183	15190	16541	60207	558090	5869831
8	18076	17725	19698	79883	650814	6759180

Table 3–9: Scatter execution time (μs) on networked workstations

$$t_{\text{scatter}}(p, d) = 56.10779 + 119.7135p + 0.017849d + 0.067336pd \ \mu\text{s} \qquad (3.12)$$

The Meiko Computing Surface also has scatter execution time linear in the number of processors. Equation 3.13 achieves an average prediction difference of 8% with a standard deviation of 12%.

$$t_{\text{scatter}}(p, d) = 684.6957 + 2797.822p - 5.97021d + 4.550472pd \ \mu s \qquad (3.13)$$

On the networked workstations, execution time is also linear in the number of processors. Equation 3.14 results in an average prediction difference of 19% and a standard deviation of 18%.

$$t_{\text{scatter}}(p, d) = 4718.983 + 335.7307p - 7.25356d + 9.410276dp \ \mu s \qquad (3.14)$$

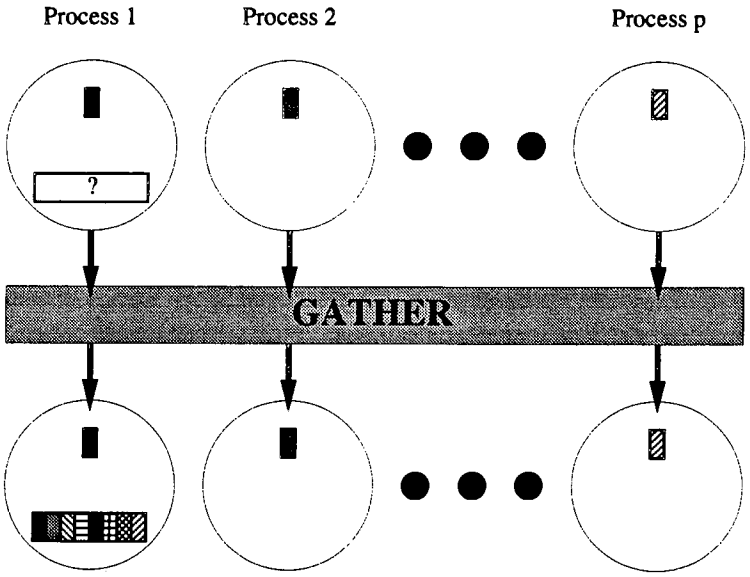


Figure 3–7: Gather

3.2.4 Gather

In a gather operation (Figure 3–7), data from each processor is concatenated onto a single gathering processor. A gather is the reverse of a scatter. In MPI, this functionality is provided by the `MPI_GATHER` operation.

Like scatter, gather can be implemented by each processor (other than the gatherer) sending a message directly to the gatherer.

Here we consider the time taken to execute:

```
      CALL MPI_GATHER(OUTBUF, SIZE,   MPI_INTEGER,
     &                INBUF,  SIZE*P, MPI_INTEGER,
     &                0, MPI_COMM_WORLD, IERROR)
```

for a variety of values of `SIZE` on various numbers of processors `P`. The execution times of a gather operation on the Cray T3D, the Meiko Computing Surface and networked workstations are shown in Tables 3–10, 3–11 and 3–12 respectively. As with scatter, gather operations of large data volumes from large numbers of processors can require more memory than is available on a single node. This occurs for some parameter values

Processors	*Message size per processor (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	226.8	227.6	239.3	354.6	1435.7	12457.9
4	246.4	248.7	261.8	421.9	2356.7	21474.7
8	285.2	288.3	309.1	643.7	4449.6	42169.0
16	364.2	367.2	403.4	1112.2	8846.4	
32	521.3	527.7	582.9	2113.4	18054.8	
64	834.5	842.3	946.6	4039.5	35924.9	
128	1456.3	1471.6	1676.3	7752.2		
256	2716.4	2756.3	3146.3	14684.0		

Table 3–10: Gather execution time (μs) on Cray T3D

on the Cray T3D and the Meiko Computing Surface, so Tables 3–10 and 3–11 are necessarily incomplete.

The execution time of gather on the T3D is linear in the number of processors. Equation 3.15 gives an average prediction difference of 10% with a standard deviation of 6%.

$$t_{\text{gather}}(p, d) = 225.3635 + 7.415866p + 0.005211d + 0.052697pd\mu\text{s} \tag{3.15}$$

On the Meiko Computing Surface, execution time is linear in the number of processors. Equation 3.16 achieves an average prediction difference of 8% with a standard deviation of 10%.

$$t_{\text{gather}}(p, d) = 4603.144 + 2097.03p - 9.26726d + 5.261446pd\mu\text{s} \tag{3.16}$$

On the networked workstations, very large variations were observed in the execution time of gather operations. A logarithmic model gives a slightly better fit than a linear approximation, but Equation 3.17 results in an average prediction difference of 38%.

Processors	*Message size per processor (integers)*				
	1	*10*	*100*	*1000*	*10000*
2	12239	12255	12416	16360	47177
4	16147	16165	16526	28220	139731
8	22283	22358	23280	54384	311201
12	31186	31342	32641	82722	
16	40793	41025	43534	112567	
20	49702	49912	52545	135732	
24	53810	54126	57188	169150	
28	64570	64924	68615	195099	
32	74522	75047	78950	227065	
36	82845	83283	87230	260923	
40	93843	94463	99376	290753	
44	97701	98535	103641	317601	
48	106775	107351	113233	347432	
52	118713	119245	125617	401750	
56	130332	130971	138367	442127	

Table 3–11: Gather execution time (μs) on Meiko Computing Surface

Processors	*Message size per processor (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	5910	5958	6350	17089	126074	1194000
3	7287	7623	8267	31374	235716	2202868
4	8836	9058	9653	41187	295077	2823539
5	10896	11111	12004	59009	442150	4188312
6	12429	12437	13833	72039	531903	5165098
7	13878	14274	15478	87142	644785	6088014
8	24361	16524	31815	122322	762070	7465391

Table 3–12: Gather execution time (μs) on networked workstations

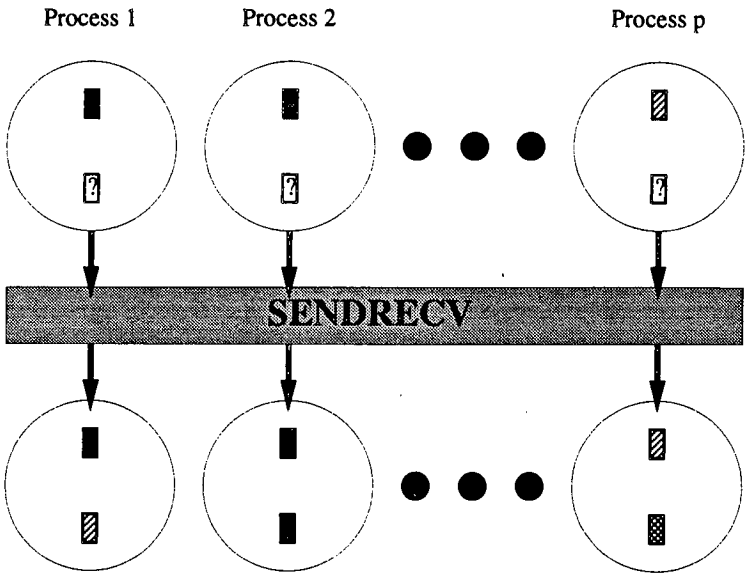


Figure 3–8: Send-receive

$$t_{\text{gather}}(p, d) = -18926.3 - 24.9043d + 11913.75 \log_2 p + 30.38917 d \log_2 p \mu\text{s} \quad (3.17)$$

3.2.5 Send-receive

In a send receive operation (Figure 3–8), provided by MPI through the `MPI_SENDRECV` routine, each process sends and receives one message.

Here we consider the time taken to execute:

```
      CALL MPI_SENDRECV(OUTBUF, SIZE, MPI_INTEGER, DEST,
     &                  0, INBUF,  SIZE, MPI_INTEGER,
     &                  MPI_ANY_SOURCE, MPI_ANY_TAG,
     &                  MPI_COMM_WORLD, STATUS, IERROR)
```

where `DEST` specifies the receiving process, for a variety of values of `SIZE` on various numbers of processors.

The execution times of send-receive operations on the Cray T3D, the Meiko Computing Surface and networked workstations are shown in Tables 3–13, 3–14 and 3–15 re-

Processors	*Message size (integers)*						
	1	*10*	*100*	*1000*	*10000*	*100000*	*1000000*
2	83.2	170.5	204.4	540.0	3827.1	35471.9	366538.7
4	84.8	172.4	201.9	545.8	3734.7	35706.9	358400.0
8	85.3	172.4	202.4	558.1	3906.9	37327.7	362334.4
16	87.1	173.1	205.3	579.1	4059.0	38661.4	377954.1
32	87.3	172.4	205.8	581.3	4080.6	38828.1	381825.9
64	87.8	173.5	203.5	567.0	3981.2	37928.1	374062.1
128	87.5	173.2	204.4	579.4	4090.1	39143.9	388795.6
256	89.9	173.5	205.9	589.3	4150.3	39779.3	401066.4

Table 3–13: Send-receive execution time (μs) on Cray T3D

spectively. Errors in low level system software prevented data being gathered for all parameter values on the Meiko Computing Surface.

The T3D performance is very close to constant for a given message size. Modelling the execution time of a barrier as the average of the measured times gives Equation 3.18, which results in an average prediction difference of 14%, with a standard deviation of 31%. For message sizes larger than 1, the average prediction difference improves to just 2%.

$$t_{\text{send-receive}}(p, d) = 195.2723 - 0.29551p + 0.367175d + 0.000142pd \ \mu s \qquad (3.18)$$

The Meiko Computing Surface has execution time linear in the number of processors. Equation 3.19 achieves an average prediction difference of 43% with a standard deviation of 25%. These differences are consistent with the observed variation in execution time.

$$t_{\text{send-receive}}(p, d) = 5838.369 - 114.467p + 1.310977d + 1.84964pd \ \mu s \qquad (3.19)$$

Processors	*Message size (integers)*					
	1	*10*	*100*	*1000*	*10000*	*100000*
2	4479	4485	4936	10102	55356	506048
4	5375	5460	5916	16426	94843	876740
8	6204	6251	6901	22224	142525	1248628
12	5903	5937	6618	27211	132076	1168063
16	4539	4593	5202	28825	153644	1353536
20	3940	4005	4637	23456	144157	1332929
24	3732	3645	4390	30529	154479	1391472
28	4242	4304	4880	31501	161603	
32	4429	4637	5683	28979	184978	1771339
36	3245	3366	3611	27703	182141	1692193
40	3842	3340	3936	27315	182127	
44	4471	4406	5196	29083	197088	
48				29099	187158	
52	3204	3140	3666	31423	172547	
56	3267	3228	4714	35461		

Table 3–14: Send-receive execution time (μs) on Meiko Computing Surface

Processors	*Message size (integers)*						
	1	*10*	*100*	*1000*	*10000*	*100000*	*1000000*
2	7070	7350	8936	33619	233760	2337999	23016601
3	7299	8018	9786	28697	305469	3209891	32333524
4	7674	7909	9622	28961	258152	2455439	36413454
5	8865	8950	10808	36541	362647	3473179	36981095
6	8308	8872	16019	34877	315323	3954539	33588071
7	8645	9237	10478	36795	332273	3109089	22829955
8	9481	9573	33989	275877	931807	5205161	48976775

Table 3–15: Send-receive execution time (μs) on networked workstations

Once again, a very large range of execution times was observed for each set of parameters on the workstation network, and the linear model in Equation 3.20 results in an average prediction difference of 60%.

$$t_{\text{send-receive}}(p, d) = -568.124p + 21.03587d + 2.792803pd\ \mu s \tag{3.20}$$

3.3 Predicting the Communication Performance of Parallel Applications

In this section we will use the performance data we have gathered for collective communication operations to predict the performance of a common parallel application structure: geometric decomposition.

We will compare two approaches to performance prediction. The first is a numerical technique, while the second relies on algebraic performance models. In both cases, we characterise a parallel application as a sequence of collective communication operations, and predict its performance as the sum of the predicted execution times of the collective operations.

Our numerical approach predicts the performance of a collective operation by linearly interpolating between the measured data values for the operation presented in Section 3.2. For example, suppose we wish to predict the execution time of a gather of 75 integers per processor on 16 processors of a Cray T3D. We know from Table 3–10 that the measured execution times on 16 processors for 10 and 100 integers are 367.2 μs and 403.4 μs respectively. We predict the execution time for 75 integers to be:

$$367.2 + \frac{75 - 10}{100 - 10} \times (403.4 - 367.2) = 393.3\mu s \tag{3.21}$$

The symbolic method calculates the execution time using the models which were derived from the measurements in Section 3.2.

Processors	*Time (s)*	*Numerical (s)*	*Difference (%)*	*Model (s)*	*Difference (%)*
1	0.243				
2	0.165	0.152	8	0.140	15
4	0.137	0.133	3	0.134	2
8	0.131	0.128	2	0.131	0
16	0.129	0.128	1	0.131	1
32	0.132	0.131	1	0.132	0
64	0.137	0.136	1	0.136	1
128	0.145	0.168	1	0.144	1

Table 3–16: Measured and predicted geometric decomposition (1024×1024, 1 iteration) execution time (s) on Cray T3D

In a geometric decomposition, an initial dataset is scattered across the processes, a number of iterations are executed, with neighbouring processes exchanging boundary values, and the final dataset is gathered from the processes.

We will concentrate on a one-dimensional geometric decomposition, in which the dataset is decomposed along a single dimension. For our studies, we will assume a $n \times 1024$ element dataset which will be decomposed along the n dimension. Each process in a p process system will therefore receive $(n \times 1024)/p$ elements in the initial scatter and send the same number in the final gather. We assume that the range of the update operation is one element, so each process will perform two send-receive operations, each of size 1024, in each of the i iterations.

For our first example, we consider the case with $n = 1024$ and $i = 1$ on the Cray T3D (Table 3–16). The numerical approach gives an average prediction difference of 4% while the model-based predictions give 3%.

Table 3–17 presents results for $n = 256$ and $i = 1$ on the Meiko Computing Surface. Here the numerical approach gives an average prediction difference of 6% while the model-based predictions give 14%. For more than 2 processors, the model achieves an average prediction difference of 7%.

Processors	*Time (s)*	*Numerical (s)*	*Difference (%)*	*Model (s)*	*Difference (%)*
1	0.354				
2	1.040	0.962	7	0.613	41
4	1.790	1.701	5	1.630	9
8	2.116	1.972	7	2.167	2
16	2.575	2.409	6	2.490	3
32	3.184	3.094	3	2.763	13

Table 3–17: Measured and predicted geometric decomposition (256×1024, 1 iteration) execution time (s) on Meiko Computing Surface

Processors	*Time (s)*	*Numerical (s)*	*Difference (%)*	*Model (s)*	*Difference (%)*
1	0.676				
2	3.351	3.179	5	2.286	32
4	3.742	3.647	3	4.415	18
8	5.067	5.209	3	4.504	11

Table 3–18: Measured and predicted geometric decomposition (256×1024, 1 iteration) execution time (s) on Networked Workstations

Running the same problem ($n = 256$, $i = 1$) on networked workstations gives the results in Table 3–18. The average prediction difference is 3% for the numerical method and 20% for the symbolic model.

Running with $n = 1024$ and $i = 1$ on workstations, we get the results shown in Table 3–19. The prediction differences for both the numerical and symbolic approaches are consistent with the smaller problem in the previous example, at 3% and 20% respectively.

Finally, Table 3–20 presents the results for a 10-iteration 1024×1024 problem on networked workstations. The average prediction differences increase slightly for both the numerical and symbolic approaches, rising to 4% and 24% respectively.

Processors	*Time (s)*	*Numerical (s)*	*Difference (%)*	*Model (s)*	*Difference (%)*
2	13.34	12.49	6	8.99	33
4	14.42	14.34	1	17.44	21
8	19.00	19.20	1	17.71	7

Table 3–19: Measured and predicted geometric decomposition (1024×1024, 1 iteration) execution time (s) on Networked Workstations

Processors	*Time (s)*	*Numerical (s)*	*Difference (%)*	*Model (s)*	*Difference (%)*
2	13.92	13.10	6	9.46	32
4	15.25	14.88	2	18.00	18
8	23.46	24.20	3	18.42	21

Table 3–20: Measured and predicted geometric decomposition (1024×1024, 10 iterations) execution time (s) on Networked Workstations

3.4 Conclusions

We have successfully predicted the execution time of the collective communication operations in parallel applications with a reasonably high degree of accuracy. Using our numerical interpolation approach we obtain very accurate predictions, while our symbolic models are less satisfactory for the Meiko Computing Surface and networked workstations. For both these platforms, inconsistencies in the measured data hindered good model fitting.

In the case of the networked workstations the inconsistencies can be attributed to a number of deficiencies in the experimental conditions. Firstly, the network was not dedicated to the experiments; it was also carrying other traffic. Secondly, the machines involved in the experiments were spread across different Ethernet segments, so networking hardware can lead to non-uniform latencies between pairs of processes on different systems.

Even with perfect experimental conditions, it is possible that the performance of a given collective operation will not take a form which is easily characterised by a single surface, as we have attempted in this chapter. It is possible for several performance regimes to exist, with discontinuities in the parameter space between these regimes. Many communication protocols exhibit this behaviour, for example with small messages piggybacking in protocol packets and achieving disproportionately high bandwidth. It would be expensive to examine the large number of possible hybrid performance functions which could be derived in this way.

Our basic cost model is also flawed, in the sense that it assumes no overlap between consecutive collective communication operations. It may well be that consecutive operations can overlap, resulting in an overestimate being made, or interact, resulting in an underestimate. Reasoning about these effects requires a detailed understanding of the implementation of the collective operations and the underlying communication fabric.

In the next chapter we use micro-analysis techniques to develop more detailed symbolic models of collective communication operations and parallel applications.

Chapter 4

Engineering the Performance of Parallel Applications

This chapter explores how symbolic performance models can be used during the development of parallel applications. In Chapter 3 we predicted the performance of parallel applications' communication patterns using coarse symbolic models of the execution time of collective communication operations. We begin this chapter by constructing more detailed symbolic models which better capture the performance of these operations. We develop models for regimes in which communication contention is minimal and maximal. In Section 4.2 we combine these expressions to produce models of various forms of domain decomposition. We relate symbolic models to each other in the subsequent sections, optimising the performance of a single implementation, comparing alternative implementations, and comparing alternative platforms. In Section 4.6 we turn our attention to more complex communication structures and discuss the difficulties of modelling their performance.

Throughout this chapter, we assume a p-processor platform in which each processor performs work at a rate of γ, with a startup latency of α and a bandwidth of β. Hence, if a processor starts to send a message of size d at time t then that message will be ready to leave the processor at time $t + \alpha$, and will arrive at the destination processor at time $t + \alpha + \frac{d}{\beta}$, in the absence of network contention. We further assume a single process is executed by each processor. Finally, we assume that the bandwidth β includes

memory-to-memory copying and data marshalling costs incurred in sending a message, as well as the network bandwidth achieved by the transport layer. Hence our platform is characterised by the parameter set $(p, \gamma, \alpha, \beta)$.

4.1 Modelling Collective Communications Performance

In this section we present micro-analysis models of broadcast, scatter, gather and send-receive. Two models are developed for each operation: one which assumes a perfect network, in which no communication contention occurs, and another which assumes a bus network in which no concurrent communications can take place.

4.1.1 Broadcast

A repeated send implementation of broadcast involves the broadcasting processor sending the data directly to each other processor in turn. In the absence of contention, the message startup costs serialise, but the transfer latencies are hidden, giving the execution time shown in Equation 4.1, where p denotes the number of processors and d is the data volume being broadcast.

$$T_{\text{broadcast}}(p, d) = (p-1)\alpha + \frac{d}{\beta} \tag{4.1}$$

If contention can occur, the execution time is given by

$$T_{\text{broadcast}}(p, d) = \alpha + (p-2)\max\left\{\alpha, \frac{d}{\beta}\right\} + \frac{d}{\beta}$$

4.1.2 Scatter

A scatter operation can be implemented as a sequence of send operations. In a contention-less environment, this implementation of scatter will serialise the message initialisations, but the message transfers will overlap, giving the execution time in Equation 4.2, where a message of size d is sent to each processor.

$$T_{\text{scatter}}(p, d) = (p-1)\alpha + \frac{d}{\beta} \tag{4.2}$$

If contention occurs in the network, then the time at which the next message leaves the processor will depend upon the message size and the network bandwidth as well as the time taken to initiate the message. A send can commence at time $\Delta t = \max\{\alpha, \frac{d}{\beta}\}$ after the previous send commences, giving Equation 4.3.

$$T_{\text{scatter}}(p, d) = \alpha + (p-2)\max\left\{\alpha, \frac{d}{\beta}\right\} + \frac{d}{\beta} \tag{4.3}$$

Note that, even when contention is possible, if $\alpha > \frac{d}{\beta}$ then we are in a regime in which startup costs dominate and execution time is given by Equation 4.2.

4.1.3 Gather

In a gather operation, a gathering process receives a single message of size d from each of the other processes. In a contention-free environment, the initiations of the sends can proceed simultaneously, as can the message transfers. This gives the model in Equation 4.4.

$$T_{\text{gather}}(p, d) = \alpha + \frac{d}{\beta} \tag{4.4}$$

In the presence of contention, the message transfers serialise, giving Equation 4.5

$$T_{\text{gather}}(p, d) = \alpha + \frac{d(p-1)}{\beta} \tag{4.5}$$

4.1.4 Send-Receive

In a contention-free environment, each processor can simultaneously initiate its send, and all of the message transfers can also proceed simultaneously, giving the execution time in Equation 4.6.

$$T_{\text{send-receive}}(p, d) = \alpha + \frac{d}{\beta} \tag{4.6}$$

In the presence of contention, the processors can still simultaneously initiate sends, but the message transfers will be serialised, giving the execution time in Equation 4.7.

$$T_{\text{send-receive}}(p, d) = \alpha + \frac{pd}{\beta} \tag{4.7}$$

4.2 Modelling Parallel Application Performance

We can combine the symbolic models of collective communication performance from Section 4.1 to produce a model of application performance. In this section we will model a number of parallel implementations of a regular grid-based computation running on a bus network. We assume that the computation iteratively updates an d-dimensional grid with n elements in each dimension, using an update operator with a range of r grid cells. This means that the value of a given cell depends on the values of all the cells within r grid cells in the previous iteration. We assume that the size of each cell's representation and the amount of work involved in updating it are constants, which we represent by s and w. Finally, we assume that i iterations are performed. Hence our workload is characterised by (d, n, r, s, w, i).

The execution time of a sequential implementation will be

$$i\frac{n^d w}{\gamma} \tag{4.8}$$

4.2.1 Basic Domain Decomposition

A parallel implementation based on a straightforward domain decomposition will divide the grid along one dimension, placing a contiguous block of the grid in each process. The implementation will first scatter the initial values of the cells to the processes, then perform a number of iterations, updating different regions of the grid in parallel and exchanging messages where necessary, before gathering the final values of the cells from the processors.

The initial scatter involves sending a message of size

$$\frac{n^d s}{p}$$

to each process. From Equation 4.3 we see that this takes time:

$$\alpha + (p-2)\max\left\{\alpha, \frac{n^d s}{\beta p}\right\} + \frac{n^d s}{\beta p} \tag{4.9}$$

Once the initial values have been scattered to the processors, we can begin iterating. In each iteration, each processor must send two to other processors and receive two edges in return. Each of these messages is of size

$$rn^{d-1}s$$

Each of these message exchanges is implemented using a send-receive operation which forces the process to wait until the data has been successfully received. If every processor performs this operation, then at time α some prn^{d-1} cells of size s have been

injected into the network for delivery. Equation 4.7 shows that the last of these will be delivered at time

$$\alpha + \frac{prn^{d-1}s}{\beta}$$

There are two edges to be exchanged by each processor, so there are two such periods, and the last processor will be ready to do its updates at time

$$2\left(\alpha + \frac{prn^{d-1}s}{\beta}\right)$$

after starting the iteration. Updating will take time

$$\frac{n^d w}{p\gamma}$$

So the total time for the iteration is

$$2\left(\alpha + \frac{prn^{d-1}s}{\beta}\right) + \frac{n^d w}{p\gamma} \tag{4.10}$$

Note that we assume that $r \leq \frac{n}{p}$. Otherwise additional communications will be necessary. Once i iterations have been completed, the final values of the cells are gathered into a single process. From Equation 4.5 we see that these will be delivered in time

$$\alpha + (p-1)\frac{n^d s}{\beta p} \tag{4.11}$$

Combining Equations 4.9, 4.10 and 4.11, we obtain the following expression for the total execution time of the basic parallel implementation:

$$\begin{aligned}&\alpha + \tfrac{n^d s}{\beta p} + (p-2)\max\left\{\alpha, \tfrac{n^d s}{\beta p}\right\} + i\left(2\left(\alpha + \tfrac{prn^{d-1}s}{\beta}\right) + \tfrac{n^d w}{p\gamma}\right) \\ &+\alpha + (p-1)\tfrac{n^d s}{\beta p}\end{aligned} \tag{4.12}$$

Figure 4–1: Cross and square stencils with $r = 2$

4.2.2 Two-Dimensional Decomposition

We can choose to decompose the dataset along two dimensions, assuming $d \geq 2$. We assume $p^{1/2}$ processors in each dimension. This continuous approximation is strictly only valid for values of p which have an integer square root. The approximation will not capture the discrete behaviour arising in practice when the two factors of p closest to $p^{1/2}$ are used. We assume that $r \leq \frac{n}{p^{1/2}}$.

The number of communications required in a two-dimensional decomposition dependences on the update operation, or more accurately, its stencil — the neighbouring cells on which the update function depends. Two two-dimensional cases are shown in Figure 4–1. If the stencil is a cross, then only nearest neighbour communication is necessary. If the stencil requires values from the corner of the square then non-nearest neighbour communication will also be required. In the one-dimensional decomposition considered above, each processor requires communication with both neighbours for both these classes of stencil.

Nearest neighbours

In a two-dimensional decomposition with nearest-neighbour communication, we have four exchanges per processor per iteration, each of size

$$\frac{n}{p^{1/2}} r n^{d-2}$$

Hence the exchanges will be complete by

$$4\alpha + \frac{rn^{d-1}s}{p^{1/2}\beta}$$

on a perfect network and by

$$4\left(\alpha + \frac{p^{1/2}rn^{d-1}s}{\beta}\right)$$

on a bus network.

Next-nearest neighbours

If the update calculation also requires values which are held on next-nearest neighbours, an additional four exchanges of

$$r^2n^{d-2}$$

cells are required, for a total iteration time of

$$4\left(\alpha + \frac{p^{1/2}rn^{d-1}s}{\beta}\right) + 4\left(\alpha + \frac{pr^2n^{d-2}s}{\beta}\right)$$

on a bus network. The expression for a perfect network is slightly more complex, since it is not clear whether the nearest neighbour or next nearest neighbours will complete first:

$$\max\left\{4\alpha + \frac{rn^{d-1}s}{p^{1/2}\beta}, 8\alpha + \frac{r^2n^{d-2}s}{\beta}\right\}$$

4.2.3 Non-blocking Implementations

Non-blocking operations allow communication to be overlapped with computation. Instead of treating each exchange in turn, we can initiate them all and then, while they are being delivered, update the cells for which we do not need information from other processors. We will concentrate on bus networks in this section.

In one dimension, this gives an iteration time of

$$2\alpha + \max\left\{\frac{\left(\frac{n}{p} - 2r\right) n^{d-1} w}{\gamma}, \frac{2prn^{d-1}s}{\beta}\right\} + \frac{2rn^{d-1}w}{\gamma} \tag{4.13}$$

The first term (2α) shows the time taken to initiate the sending of two messages, the edges which must be sent to neighbouring processes. The max term combines the time taken to update the interior cells of this process's block

$$\frac{\left(\frac{n}{p} - 2r\right) n^{d-1} w}{\gamma}$$

and the time taken for the network to deliver the edge cells that are being sent by all the processes

$$\frac{2prn^{d-1}s}{\beta}$$

Once whichever of these two takes longer has completed, we know that the neighbouring processes' edge cells have arrived, and that the processor is free to update this block's edge cells. The time taken to do this is represented by the final term

$$\frac{2rn^{d-1}w}{\gamma}$$

The same form of expression is obtained for other numbers of dimensions too. For a two-dimensional decomposition with nearest neighbour stencil, we have

$$4\alpha + \max\left\{\frac{\left(\frac{n}{p^{1/2}} - 2r\right)^2 n^{d-2}w}{\gamma}, \frac{4p^{1/2}rn^{d-1}s}{\beta}\right\} + \left(\frac{4rn}{p^{1/2}} - 4r^2\right)\frac{n^{d-2}w}{\gamma}$$

If next-nearest neighbour communication is required, our iteration time becomes:

$$8\alpha + \max\left\{\frac{\left(\frac{n}{p^{1/2}} - 2r\right)^2 n^{d-2}w}{\gamma}, \frac{4p^{1/2}rn^{d-1}s}{\beta} + \frac{4pr^2n^{d-2}s}{\beta}\right\}$$
$$+ \left(\frac{4rn}{p^{1/2}} - 4r^2\right)\frac{n^{d-2}w}{\gamma}$$

4.3 Optimising the Performance of a Particular System

The programmer can use a performance model to optimise the parameters of a particular implementation with respect to a particular metric. For example, we can produce expressions for the speedup of the basic parallel implementation (Equation 4.12) relative to the sequential implementation (Equation 4.8). Using case analysis to remove the max from Equation 4.12 we obtain

$$\text{Speedup}(p) = \frac{in^d w\beta p}{\alpha\beta p^2\gamma n + 2i\alpha\beta p\gamma n + 2ip^2rn^d s\gamma + in^{d+1}w\beta + n^{d+1}s\gamma p}$$

where $\alpha \geq \frac{n^d s}{\beta p}$, and

$$\text{Speedup}(p) = \frac{in^{d+1}w\beta p}{2(\alpha\beta p\gamma n - n^{d+1}s\gamma + n^{d+1}s\gamma p + i\alpha\beta p\gamma n + ip^2rn^d s\gamma) + in^{d+1}w\beta}$$

where $\alpha \leq \frac{n^d s}{\beta p}$. In both cases the expressions are of the form:

$$\text{Speedup}(p) = \frac{c_1 p}{c_2 p^2 + c_3 p + c_4}$$

These expressions show that, ultimately, increasing p will reduce Speedup(p), but there may be an initial regime in which it increases with increasing p, depending on the relative values of c_1, c_2, c_3 and c_4. In this case, the programmer can look out for the point at which additional processors do not make an acceptable impact on performance. Given values for the parameters of the model, this point can be predicted.

For other parallel application structures there can be an optimal number of processors, which maximises the tradeoff between additional communication costs and reduced workload per processor. Using less, or more, processors results in higher execution time. A symbolic model could allow the programmer to determine this optimal number of processors.

Example 4.3.1 *Consider a reaction-diffusion simulation on a two-dimensional mesh ($d = 2$), of size 1000 elements in each dimension ($n = 1000$). The simulation involves nearest neighbour interactions with immediately adjacent cells ($r = 1$). Let us suppose that the representation of each cell's state requires 16 bytes of storage ($s = 16$), that updating each cell in a given iteration requires 20 floating point operations ($w = 20$), and that we are interested in running this simulation for 100 iterations ($i = 100$).*

Consider a cluster of workstations on an Ethernet. Suppose that each workstation is capable of 2×10^7 floating point operations per second, and that we have up to 10 such workstations available ($p \in \{1 \ldots 10\}$. In this environment, α is typically 10^{-3}s, while β is 10^6 byte/s.

We can quickly determine that we are in the second of the two regimes, since the first requires:

$$10^{-3} \geq \frac{10^6 \times 16}{10^6 \times p}$$

i.e.

$$p \geq 16,000$$

which is outwith the region of parameter space in which we are interested.

By substitution, we can simplify the expression for speedup in the second regime to

$$\text{Speedup}(p) = \frac{p}{0.032 \times p^2 + 0.32202 \times p + 0.68} \tag{4.14}$$

For sufficiently large p, the p^2 component of the denominator will dominate and the speedup will fall as p increases. This reflects the network bandwidth becoming the bottleneck and the increasing network contention as the increasing number of processors requires more data to be communicated in each iteration.

By differentiating Equation 4.14, and solving for p when the derivative is zero, we can determine the point at which speedup is maximal.

$$\frac{d}{dp}\text{Speedup}(p)$$

is given by

$$\frac{0.68 - 0.032p^2}{(0.032p^2 + 0.32202p + 0.68)^2}$$

which has a zero at $p \approx 4.6$, suggesting that the maximum speedup will be achieved with $p = 4$ or $p = 5$. This is borne out by the tabulated speedup figures shown below, which show a maximal speedup at $p = 5$.

p	Speedup(p)
1	*0.97*
2	*1.38*
3	*1.55*
4	*1.61*
5	*1.62*
6	*1.59*
7	*1.55*
8	*1.51*
9	*1.46*
10	*1.41*

4.4 Selecting the Better of Two Implementations

In Section 4.2.3, a model for a non-blocking implementation of the one-dimensional problem was presented. We can determine the regimes of parameter space in which this outperforms the basic implementation by constructing an inequality from the models of the two implementations' execution times. The scatter and gather phases will take identical amounts of time, so we can concentrate on the iteration times for the two implementations (Equations 4.10 and 4.13). Case analysis is required to remove the max operators from the expressions. We find that, for the non-blocking implementation to execute more quickly than the basic implementation, we require:

$$\frac{2prn^{d-1}s}{\beta} > 0$$

(which is always true for positive parameter values) when

$$(n - 2rp)w\beta \geq 2p^2rs\gamma$$

or

$$p < \frac{n}{2r}$$

when

$$(n - 2rp)w\beta \leq 2p^2rs\gamma$$

Given characterisations of the workload and the application, these expressions can be used to select the more appropriate implementation.

Example 4.4.1 *Using the same application and platform as in Example 4.3.1, we can see that we are in the second regime, since:*

$$(\frac{n}{p} - 2r)w\beta \leq 2prs\gamma$$

requires

$$125 - 250p < 4p^2$$

which is true for all $p \geq 1$.

In this regime, the non-blocking implementation is faster if

$$p < \frac{n}{2r}$$

i.e.

$$p < 500$$

So for $p \in \{1, \ldots, 10\}$, *the non-blocking implementation is faster in our case. This is borne out by the tabulated iteration times of the basic and non-blocking parallel implementations below.*

p	*Basic time, s*	*Non-blocking time, s*
1	*1.034*	*1.002*
2	*0.566*	*0.502*
3	*0.431*	*0.335*
4	*0.380*	*0.252*
5	*0.362*	*0.202*
6	*0.361*	*0.196*
7	*0.369*	*0.228*
8	*0.383*	*0.260*
9	*0.401*	*0.292*
10	*0.422*	*0.324*

The non-blocking implementation is faster because it overlaps most of its computation with the communication, which is the dominant component of the iteration time. At $p = 500$*, we reach the point where the unoverlapped computation in the non-blocking implementation is the same as the computation in the basic implementation, so the non-blocking implementation no longer has a performance advantage. However, we can see from the table that optimal speedup is obtained with* $p = 6$*, a relatively modest number of processors.*

4.5 The Effect of Increased Resources

By combining Equations 4.9, 4.11 and 4.13, we can show that a non-blocking implementation of a one-dimensional decomposition has execution time:

$$\begin{aligned} &\alpha + \tfrac{n^d s}{\beta p} + (p-2)\max\left\{\alpha, \tfrac{n^d s}{\beta p}\right\} \\ &+i\left(2\alpha + \max\left\{\tfrac{(n-2r)n^{d-1}w}{\gamma}, \tfrac{2prn^{d-1}s}{\beta}\right\} + \tfrac{2rn^{d-1}w}{\gamma}\right) \\ &+\alpha + (p-1)\tfrac{n^d s}{\beta p} \end{aligned} \tag{4.15}$$

Suppose that we are faced with a choice of doubling the bandwidth β or doubling the processing rate γ. We can establish the regimes of parameter space in which the former is the better choice by comparing two versions of Equation 4.15 — one with γ replaced by 2γ, and the other with β replaced by 2β. The resulting case analysis gives rise to three regions of parameter space, each with an associated symbolic condition which must be satisfied for the first system to outperform the second. Comparison with the original implementation would also be necessary, in order to determine the performance improvement offered by the new implementations.

4.6 More Complex Structures

The models developed in this chapter have been comparatively straightforward in terms of the potential for contention and interaction between processes. More complex patterns of interactions can arise in quite simple systems. For example, tree implementations of collective communications or applications employing dynamic load-balancing.

4.6.1 Collective Communications

In a b-ary tree implementation of a scatter operation, there are $\lfloor \log_b p \rfloor$ layers of communication. In layer $i \in \{1 \ldots \lfloor \log_b p \rfloor - 1\}$ there are b^i communications of size $\sum_{k=1}^{\lfloor \log_b p \rfloor + 1 - i} b^{k-1} d = \left(\frac{b^{(\lfloor \log_b p \rfloor - i + 1)} - 1}{b-1} \right) d$. In layer $i = \lfloor \log_b p \rfloor$ there is at least one communication and at most b^i communications of size d.

In a contention-free environment, we can lower bound the execution time of scatter by

$$T_{\text{scatter}}(p,d) \geq \sum_{i=1}^{\lfloor \log_b p \rfloor - 1} \left(\alpha + \frac{1}{\beta} \frac{b^{(\lfloor \log_b p \rfloor - i + 1)} - 1}{b-1} d \right) + \alpha + \frac{d}{\beta} \qquad (4.16)$$

and upper bound it by

$$\begin{aligned} T_{\text{scatter}}(p,d) &\leq \sum_{i=1}^{\lfloor \log_b p \rfloor} \left(b\alpha + \frac{1}{\beta} \frac{b^{(\lfloor \log_b p \rfloor - i + 1)} - 1}{b-1} d \right) \\ &= \lfloor \log_b p \rfloor b\alpha + \frac{1}{\beta} \left(\frac{b^{\lfloor \log_b p \rfloor + 1} - b\lfloor \log_b p \rfloor - b + \lfloor \log_b p \rfloor}{(b-1)^2} \right) d \end{aligned}$$

which we can approximate by

$$\lfloor \log_b p \rfloor b\alpha + \frac{1}{\beta} \left(\frac{bp - b\lfloor \log_b p \rfloor - b + \lfloor \log_b p \rfloor}{(b-1)^2} \right) d$$

The situation becomes much more complex if contention is possible. We can notice that $p-1$ processes must receive a message of size d, therefore the execution time of the gather has to be at least

$$\alpha + \frac{(p-1)d}{\beta} \tag{4.17}$$

We can combine Equations 4.16 and 4.17 to obtain a new lower bound, but we cannot produce an upper bound without taking account of the pattern of interactions in much more detail. It is not clear that the network is saturated, and that there are not idle periods while the network waits for messages to be injected. Hence our simple bottleneck calculation cannot provide a reliable upper bound.

If we are in a regime where the message transfers serialise, and only the initial startup is a visible component of the execution time, then $T_{\text{scatter}}(p,d)$ is given by:

$$\begin{aligned} & \alpha + \sum_{i=1}^{\lfloor \log_b p \rfloor} \left(\frac{1}{\beta} \frac{b^{\lfloor \log_b p \rfloor - i + 1} - a}{b-1} b^i d \right) \\ = \; & \alpha + \left(\frac{b^{\lfloor \log_b p \rfloor + 2} \lfloor \log_b p \rfloor - b^{\lfloor \log_b p \rfloor + 1} \lfloor \log_b p \rfloor - b^{\lfloor \log_b p \rfloor + 1} + b}{\beta (b-1)^2} \right) d \end{aligned}$$

which we can approximate by

$$\alpha + \frac{1}{\beta} \left(\frac{b^2 p \lfloor \log_b p \rfloor + bp \lfloor \log_b p \rfloor - bp + b}{(b-1)^2} \right) d$$

4.6.2 Other Parallel Application Structures

In a task farm, work is allocated to processors dynamically, and the behaviour of the system is highly dependent upon the timing of interactions in the network. A task farm consists of a single master process and a number of worker processes. All of the workers

send a message to the master requesting a task. In a contention-free environment these messages all arrive at time

$$\alpha + \frac{q}{\beta}$$

where q is the size of a request message. The master subsequently responds to the $p-1$ requests which it has received, and worker process $i \in \{1, \ldots, p-1\}$ receives its task at

$$\alpha + \frac{q}{\beta} + i\alpha + \frac{s}{\beta}$$

where s is the size of the task. The worker then takes time t to perform the task before sending back a combined result and request for further work, which we assume is of size q. This second request arrives at the master at time:

$$(i+2)\alpha + 2\frac{q}{\beta} + \frac{s}{\beta} + t$$

The time at which the master will respond will depend upon the other requests to which it is responding. These are, of course, a function of the other worker process's behaviour. A task farm is often used when the time t to perform a task is not a constant, and this complicates the analysis further. Finally, if contention can occur, it is difficult to predict the way in which request and task messages may interact due to the complexity of the system's behaviour.

4.7 A Framework for Managing Complexity

In this chapter we have seen a number of ways in which symbolic models of execution time can be used to support design decisions, helping the developer optimise a particular implementation, select one of a number of alternative implementations, or choose one of a number of platforms. However, developing such models for complex communication patterns which arise routinely in parallel applications is a demanding task.

For both the collective communication example and the task farm example in Section 4.6, we have good knowledge of the sequence of actions which each process will perform, but we struggle to keep track of the complex interactions between these processes which determine the performance of the system. In the next chapter we will develop a framework in which we can reason formally about the performance of parallel systems. This formal approach will not simplify the analysis process itself, but it will provide a framework in which we can manage the complexity of interaction patterns, and allow us to derive performance measures for models which are not amenable to informal analysis.

Chapter 5

Reasoning Formally About the Execution Time of Concurrent Systems

In Chapter 4 we saw a number of difficulties which can arise in developing symbolic expressions for the execution time of concurrent systems. These difficulties were primarily due to the complexity of interactions between components of a parallel system giving rise to behaviours which are difficult to reason about informally.

Process calculi such as CSP [Hoare, 1985], CCS [Milner, 1989], ACP [Bergstra and Klop, 1985] and their derivatives have been used to provide a formal framework for reasoning about the behaviour of concurrent systems. A number of process calculi have been augmented with a notion of time and used to reason about aspects of the temporal behaviour of concurrent systems. One such calculus is Chen's Timed CCS [Chen, 1993]. In this chapter we explore how models of (execution time) performance can be derived from descriptions of concurrent systems in a timed process calculus, derived from Timed CCS.

We begin by presenting Timed CCS. We introduce the notion of eagerness in Section 5.2, then define Eager Timed CCS and relate it to Timed CCS. In Section 5.3 we provide a quantitative metric for the execution time performance of Eager Timed CCS agents and discuss the metric's interaction with behavioural reasoning. We consider the automation of the analysis process in Section 5.3.3 and present a number of example

analyses. The chapter closes with a discussion of related work and issues for further research.

5.1 Timed CCS

Timed CCS [Chen, 1993] is an extension of Milner's Calculus of Communicating Systems (CCS) [Milner, 1980; Milner, 1989] which includes time.

To define Timed CCS, Chen presupposes a countable set Λ of names of atomic actions ranged over by a, b and not containing τ. Let $Act = \Lambda \cup \{\tau\}$, ranged over by α, β. As in CCS, Λ can be partitioned into Γ, the set of names, and $\overline{\Gamma} = \{\overline{a} | a \in \Gamma\}$, the set of co-names, with the provision that $\overline{\overline{a}} = a$. Actions a and $\overline{a}$ are called complementary actions and form the basis of communications in our language, analogous to CCS. We also presuppose a countable set V_p of process variables, ranged over by X, Y, and a countable set $V_\mathcal{T}$ of time variables, ranged over by t, s, r. Let the time domain be $(\mathcal{T} \cup \{\infty\}, \leq)$ where $\leq$ is a linear order over $\mathcal{T}$ and $0 \in \mathcal{T}$ represents the starting time. Note that no assumptions are made about the underlying nature of time, allowing $\mathcal{T}$, for example, to be $\mathbb{N}$ (the set of natural numbers), $\mathbb{Q}^{\geq 0}$ (the set of non-negative rationals) or $\mathbb{R}^{\geq 0}$ (the set of non-negative reals). Timed CCS time expressions, ranged over by e, f, g, are defined as follows:

Definition 5.1.1 *The set $\mathcal{E}_\mathcal{T}$ of time expressions is defined as follows:*

1. *For every $u \in \mathcal{T}$ and $t \in V_\mathcal{T}$, $u \in \mathcal{E}_\mathcal{T}$ and $t \in \mathcal{E}_\mathcal{T}$.*

2. *For every $u \in \mathcal{T}$ and $e \in \mathcal{E}_\mathcal{T}$, $u \times e \in \mathcal{E}_\mathcal{T}$.*

3. *If $e, f \in \mathcal{E}_\mathcal{T}$, then $e + f$, $e - f$, $e \ominus f$, $\max(e, f)$, $\min(e, f)$ are all in $\mathcal{E}_\mathcal{T}$, where*

$$e \ominus f = \begin{cases} 0 & (\text{if } e < f) \\ e - f & (\text{otherwise}) \end{cases}$$

By convention, for any $e \in \mathcal{E}_\mathcal{T}$ *we have* $e \leq \infty$, $\infty + e = \infty$, $\infty - e = \infty$, $\max(e, \infty) = \infty$, $\min(e, \infty) = e$.

The process expressions of Timed CCS, ranged over by E, F, are defined by the following BNF expression:

$$E ::= X \mid nil \mid \alpha@t_e^{e'}.E \mid E + F \mid E|F \mid E\backslash a \mid E[f] \mid \mu X.E$$

where $f : Act \to Act$ is a relabelling function which satisfies $\overline{f(a)} = f(\overline{a})$ and $f(\tau) = \tau$, and e and e' are time expressions or e' is the infinite time ∞.

Process nil cannot perform any action but may idle for any period of time.

Prefix $\alpha@t_e^{e'}.E$ (written $\alpha_e^{e'}(t).E$ in Chen's original notation) represents the process which performs action α at some relative time between e and e' (inclusive) from now, where e and e' are called the lower bound and upper bound of action α respectively. Note that only the upper bound is permitted to be the infinite time ∞. Any occurrences of time variable t in E refer to the happening time of action α. After α happens, the process evolves to $E\{u/t\}$ where u is the time at which α happens and $E\{u/t\}$ is the result of substituting all free occurrences of t by u in E. Where t does not occur in E, we will omit t, writing $\alpha_e^{e'}.E$. When the upper bound e' is ∞ we will omit it and write $\alpha@t_e.E$. Where appropriate, we will omit both the time variable and the infinite upper bound, writing $\alpha_e.E$.

Summation $E + F$ represents choice between processes E and F. The choice is made at the time of the first action of E or F, or at the time when only one process can idle, in which case the process which cannot delay is dropped from the future computation. Clearly this choice is deterministic with respect to time proceeding.

Process $E|F$ represents the parallel composition of processes E and F. Each of them may perform actions independently, or they may synchronise on complementary actions which represent communications between them, analogous to CCS. Parallel composition is synchronous with respect to time proceeding, i.e. the parallel composition $E|F$ of the processes E and F can delay for time u only when both E and F can.

The process $E[f]$ represents the relabelling of E according to the relabelling function $f : Act \longrightarrow Act$ which satisfies $\overline{f(a)} = f(\overline{a})$ and $f(\tau) = \tau$. $E[f]$ behaves as the process E, except that whenever E performs an action α, $E[f]$ performs the action $f(\alpha)$.

The process $E\backslash L$, where $L \subseteq \Lambda$, represents the restriction of E which can perform any action α of which E is capable, except where $\alpha \in L$ or $\overline{\alpha} \in L$.

The recursive operator μX in $\mu X.E$ binds all free occurrences of process variable X in E. This gives the usual notions of free and bound occurrences of process variables.

We use $fv_p(E)$ and $fv_t(E)$ to represent the set of all free process and time variables occurring in E. We identify those process expressions which are the same up to changes of bound process and time variables. We say a process expression E is closed with respect to process variables (or time variables) if there are no free occurrences of process variables (or time variables) in E, i.e. $fv_p(E) = \emptyset$ (or $fv_t(E) = \emptyset$). An agent is a process expression which is closed with respect to process variables and time variables. We let $\mathcal{P}$ represent the set of agents which is ranged over by P, Q, R.

We say a process P is weakly guarded if every process variable of P is weakly guarded in P, where X is weakly guarded in P if every occurrence of X is in some sub-term of the form $\alpha@t_e^{e'}.E$ of P.

Chen gives an operational semantics for Timed CCS using the general notion of a labelled transition system:

$$(S, \{\xrightarrow{t} : t \in T\})$$

which consists of a set S of *states*, a set T of *transition labels* and a *transition relation* $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$. In the transition system of Timed CCS, Chen takes S to be $\mathcal{P}$, the set of agents, and T to be $(Act \cup \{\epsilon\}) \times \mathcal{T}$, the set of pairs of actions and times (recall that $\epsilon \notin Act$). Chen writes $P \xrightarrow{\alpha}_u P'$ in place of $(P, P') \in \xrightarrow{(\alpha,u)}$, where $\alpha \in Act$ and $u \in \mathcal{T}$ and $P \longrightarrow_u P'$ in place of $(P, P') \in \xrightarrow{(\epsilon,u)}$.

The understanding of the transition $P \xrightarrow{\alpha}_u P'$ is that agent P performs action α at time u and then evolves to P'. The transition $P \longrightarrow_u P'$ means that the agent P idles for time u without any action and then evolves to P'.

Chen relies on Moller and Tofts' definition of maximal delay time of processes before any action:

Definition 5.1.2 $|\ |_{\mathcal{T}} : \mathcal{P} \to \mathcal{T}$ *gives the maximum time for which a Timed CCS agent can delay before performing an action.*

$$
\begin{aligned}
|nil|_{\mathcal{T}} &= \infty && (1)\\
|\alpha@t_e^{e'}.P|_{\mathcal{T}} &= e' && (2)\\
|P+Q|_{\mathcal{T}} &= \max(|P|_{\mathcal{T}}, |Q|_{\mathcal{T}}) && (3)\\
|P|Q|_{\mathcal{T}} &= \min(|P|_{\mathcal{T}}, |Q|_{\mathcal{T}}) && (4)\\
|P\backslash L|_{\mathcal{T}} &= |P|_{\mathcal{T}} && (5)\\
|P[f]|_{\mathcal{T}} &= |P|_{\mathcal{T}} && (6)\\
|\mu X.P|_{\mathcal{T}} &= |P\{\mu X.P/X\}|_{\mathcal{T}} && (7)
\end{aligned}
$$

Note that the maximal delay $|P|_{\mathcal{T}}$ of agent P is well-defined only when P is weakly guarded.

All transition rules are presented in Table 5–1. The rules are read as follows: if the transition or transitions above the inference line can be inferred, then we can infer the transition below the line. The operational semantics of the language is given by the least transition relations $\xrightarrow{\alpha}_u$ and $\longrightarrow_u$, where $\alpha \in Act$ and $u \in \mathcal{T}$, defined in Table 5–1.

Chen shows that the following theorems hold:

Theorem 5.1.3 *If $P \longrightarrow_u P'$ and $P \longrightarrow_u P''$ then $P' \equiv P''$. This shows that the passing of time is deterministic.*

Theorem 5.1.4 *If $P \longrightarrow_u P'$ and $P' \longrightarrow_v P''$ then $P \longrightarrow_{u+v} P''$. This shows that time is continuous for delay transitions.*

Theorem 5.1.5 *If $P \longrightarrow_u P'$ and $P' \xrightarrow{\alpha}_v P''$ then $P \xrightarrow{\alpha}_{u+v} P''$. This shows that time is continuous for a delay transition followed by an action transition.*

1 $\dfrac{}{\alpha@t_v^{v'}.E \xrightarrow{\alpha}_u E\{u/t\}}(v \leq u \leq v')$

2 $\dfrac{}{\alpha@t_v^{v'}.E \longrightarrow_u \alpha@t_{v\ominus u}^{v'-u}.(E\{u+t/t\})}(u \leq v')$

3 $\dfrac{}{nil \longrightarrow_u nil}$

4 $\dfrac{P \longrightarrow_u P' \quad Q \longrightarrow_u Q'}{P+Q \longrightarrow_u P'+Q'}$

5 $\dfrac{P \longrightarrow_u P'}{P+Q \longrightarrow_u P'}(|Q|_\tau < u)$

6 $\dfrac{Q \longrightarrow_u Q'}{P+Q \longrightarrow_u Q'}(|P|_\tau < u)$

7 $\dfrac{P \xrightarrow{\alpha}_u P'}{P+Q \xrightarrow{\alpha}_u P'}$

8 $\dfrac{Q \xrightarrow{\alpha}_u Q'}{P+Q \xrightarrow{\alpha}_u Q'}$

9 $\dfrac{P \longrightarrow_u P' \quad Q \longrightarrow_u Q'}{P|Q \longrightarrow_u P'|Q'}$

10 $\dfrac{P \xrightarrow{\alpha}_u P' \quad Q \xrightarrow{\overline{\alpha}}_u Q'}{P|Q \xrightarrow{\tau}_u P'|Q'}$

11 $\dfrac{P \xrightarrow{\alpha}_u P' \quad Q \longrightarrow_u Q'}{P|Q \xrightarrow{\alpha}_u P'|Q'}$

12 $\dfrac{P \longrightarrow_u P' \quad Q \xrightarrow{\alpha}_u Q'}{P|Q \xrightarrow{\alpha}_u P'|Q'}$

13 $\dfrac{P \longrightarrow_u P'}{P\backslash L \longrightarrow_u P'\backslash L}$

14 $\dfrac{P \xrightarrow{\alpha}_u P'}{P\backslash L \xrightarrow{\alpha}_u P'\backslash L}(\alpha \notin L \wedge \overline{\alpha} \notin L)$

15 $\dfrac{P \longrightarrow_u P'}{P[f] \longrightarrow_u P'[f]}$

16 $\dfrac{P \xrightarrow{\alpha}_u P'}{P[f] \xrightarrow{f(\alpha)}_u P'[f]}$

17 $\dfrac{E\{\mu X.E/X\} \longrightarrow_u P}{\mu X.E \longrightarrow_u P}$

18 $\dfrac{E\{\mu X.E/X\} \xrightarrow{\alpha}_u P}{\mu X.E \xrightarrow{\alpha}_u P}$

Table 5–1: Operational Rules for Timed CCS

In Theorems 5.1.4 and 5.1.5, Chen's use of "continuous" is confusing, since the theorem holds for a dense, continuous time domain or a discrete, non-continuous time domain.

Theorem 5.1.6 *If* $P \xrightarrow{\alpha}_u P'$ *then* $P \longrightarrow_u P''$ *and* $P'' \xrightarrow{\alpha}_0 P'$ *for some* P''. *This shows that actions are instantaneous.*

5.2 Eager Timed CCS

Timed CCS agents do not have to perform actions as soon as they become possible. With infinite upper bounds on actions, agents can delay actions indefinitely. For example, the agent $a@t_3^\infty.nil$ can evolve to nil through the transition $\xrightarrow{a}_3$ or through the transition $\xrightarrow{a}_{1000000}$ or through any transition $\xrightarrow{a}_u$ where $u \geq 3$. When we are considering the execution time of agents this ability to inject delays into the execution time is unnatural.

In this section we will define Eager Timed CCS, a variant of Timed CCS in which agents must perform actions as soon as possible. In our treatment we will assume that all upper bounds on actions are ∞.

We will give a semantics for Eager Timed CCS in the form of a transition system

$$(S, \{\stackrel{t}{\Rightarrow} : t \in T\})$$

in which the set S of states is $\mathcal{P}$ the set of Eager Timed CCS agents, and the set T of transition labels is $(Act \times \mathcal{T})$, the set of pairs of actions and times. As before, we will write $P \stackrel{\alpha}{\Longrightarrow}_u P'$ in place of $(P, P') \in \stackrel{(\alpha,u)}{\Rightarrow}$ where $\alpha \in Act$ and $u \in \mathcal{T}$. The understanding of this transition is that P performs action α at time u and then evolves to P'.

A key component of our definition will be a function $\mathcal{I}(P, \alpha)$ which will give the times at which an agent P can perform α as its next action. Before defining $\mathcal{I}$ we introduce a notational convenience:

Definition 5.2.1 *We define a notational shorthand for intervals of time:*

$$[u_1, u_2] = \{u | u_1 \leq u \leq u_2\}$$

We are now in a position to define $\mathcal{I}$, the intervals in which a process can perform a given action as its next action.

Definition 5.2.2 $\mathcal{I} : \mathcal{P} \times Act \to \mathbb{F}\ (\mathcal{T} \times \mathcal{T})$ *gives the intervals in which a process may possibly perform a certain action as its next action. Note that* $\mathcal{I}$ *takes no account of preemption or eagerness of actions other than insisting that explicit* τ *actions must occur as soon as possible. Recall that* a *and* b *range over* Λ*, while* α *and* β *range over* $Act = \Lambda \cup \{\tau\}$.

$$\begin{aligned}
\mathcal{I}(nil, \alpha) &= \emptyset && (1)\\
\mathcal{I}(a_v.P, a) &= \{[v, \infty]\} && (2)\\
\mathcal{I}(\tau_v.P, \tau) &= \{[v, v]\} && (3)\\
\mathcal{I}(\alpha_v.P, \beta) &= \emptyset \quad (\text{if } \alpha \neq \beta) && (4)\\
\mathcal{I}(P + Q, \alpha) &= (\mathcal{I}(P, \alpha) \cup \mathcal{I}(Q, \alpha)) && (5)\\
\mathcal{I}(P|Q, a) &= (\mathcal{I}(P, a) \cup \mathcal{I}(Q, a)) && (6)\\
\mathcal{I}(P|Q, \tau) &= \mathcal{I}(P, \tau) && (7)\\
&\cup\ \mathcal{I}(Q, \tau)\\
&\cup\ \{[u, u] |\ \exists b \in \Lambda.\\
&\qquad \exists [u_1, u_2] \in \mathcal{I}(P, b).\\
&\qquad \exists [u_3, u_4] \in \mathcal{I}(Q, \bar{b}).\\
&\qquad ([u_1, u_2] \cap [u_3, u_4] \neq \emptyset)\\
&\qquad \wedge u = \max(u_1, u_3)\}\\
\mathcal{I}(P\backslash L, a) &= \{[u_1, u_2] | [u_1, u_2] \in \mathcal{I}(P, a) \wedge a \notin L \wedge \bar{a} \notin L\} && (8)\\
\mathcal{I}(P\backslash L, \tau) &= \mathcal{I}(P, \tau) && (9)\\
\mathcal{I}(P[f], \alpha) &= \{[u_1, u_2] | \exists \beta \in Act. f(\beta) = \alpha \wedge [u_1, u_2] \in \mathcal{I}(P, \beta)\} && (10)\\
\mathcal{I}(\mu X.P, \alpha) &= \mathcal{I}(P\{\mu X.P/X\}, \alpha) && (11)
\end{aligned}$$

We can generalise from the definition which $\mathcal{I}$ provides of the time at which an agent can perform a certain action. The function δ gives the set of actions which an agent can perform as its next action and the times at which it can perform them.

Definition 5.2.3 *The function* $\delta : \mathcal{P} \rightarrow \mathbb{F}\ (Act \times \mathbb{F}\ (\mathcal{T} \times \mathcal{T}))$ *gives the set of all possible interval-action pairs for the given weakly-guarded process's next action.*

$$\delta(P) = \bigcup_{\alpha \in Act} \{(\alpha, [u_1, u_2]) | [u_1, u_2] \in \mathcal{I}(P, \alpha)\}$$

We want to allow an agent P to perform the earliest transitions in $\delta(P)$, and we now define $\Delta(P)$.

Definition 5.2.4 *The function* $\Delta : \mathcal{P} \rightarrow \mathbb{F}\ (Act \times \mathcal{T})$ *is the set of times and actions of the earliest possible actions which the given weakly-guarded process can perform.*

$$\Delta(P) = \{(\alpha, u_1) | \exists (\alpha, [u_1, u_2]) \in \delta(P). \forall (\beta, [u_3, u_4]) \in \delta(P). u_1 \leq u_3\}$$

So we can allow all transitions $P \stackrel{\alpha}{\Longrightarrow}_u$ such that $(\alpha, u) \in \Delta(P)$. However, we need to define P' such that $P \stackrel{\alpha}{\Longrightarrow}_u P'$. In general there will be a set of possibilities, which we define using two operators Delay and Action.

Definition 5.2.5 Delay : $\mathcal{P} \times \mathcal{T} \rightarrow \mathcal{P}$ *gives the agent resulting from delaying the given weakly-guarded agent by the specified period of time.*

$$\begin{aligned}
\text{Delay}(nil, u) &= nil && (1)\\
\text{Delay}(\alpha@t_v.P, u) &= \alpha@t_{v \ominus u}.P\{u + t/t\} && (2)\\
\text{Delay}(P + Q, u) &= \text{Delay}(P, u) + \text{Delay}(Q, u) && (3)\\
\text{Delay}(P|Q, u) &= \text{Delay}(P, u)|\text{Delay}(Q, u) && (4)\\
\text{Delay}(P\backslash L, u) &= \text{Delay}(P, u)\backslash L && (5)\\
\text{Delay}(P[f], u) &= \text{Delay}(P, u)[f] && (6)\\
\text{Delay}(\mu X.P, u) &= \text{Delay}(P\{\mu X.P/X\}, u) && (7)
\end{aligned}$$

Note that $\mu X.P$ *must be weakly guarded, and that* $\text{Delay}(P, t)$ *assumes that no action can take place before* t *has elapsed.*

The Action operator returns a multi-set of processes. We use the notation $\mathbb{M}\ (S)$ to denote the set of finite multi-sets containing elements of S.

Definition 5.2.6 Action : $\mathcal{P} \times Act \rightarrow \mathbb{M}(\mathcal{P})$ *gives the set of derivatives which can be reached from the given weakly-guarded agent by performing the specified action immediately.*

$$
\begin{array}{lcll}
\text{Action}(nil, \alpha) & = & \emptyset & (1) \\
\text{Action}(\alpha@t_v.P, \alpha) & = & \{\!|P\{0/t\}|\!\} \quad (\text{if } v = 0) & (2) \\
\text{Action}(\alpha@t_v.P, \alpha) & = & \emptyset \quad (\text{if } 0 < v) & (3) \\
\text{Action}(P + Q, \alpha) & = & \text{Action}(P, \alpha) \cup \text{Action}(Q, \alpha) & (4) \\
\text{Action}(P|Q, a) & = & \{\!|P'|Q \mid P' \in \text{Action}(P, a)|\!\} & (5) \\
 & \uplus & \{\!|P|Q' \mid Q' \in \text{Action}(Q, a)|\!\} & \\
\text{Action}(P|Q, \tau) & = & \{\!|P'|Q \mid P' \in \text{Action}(P, \tau)|\!\} & (6) \\
 & \uplus & \{\!|P|Q' \mid Q' \in \text{Action}(Q, \tau)|\!\} & \\
 & \uplus & \{\!|P'|Q' \mid \exists b \in \Lambda. & \\
 & & \quad P' \in \text{Action}(P, b) & \\
 & & \quad \wedge Q' \in \text{Action}(Q, \overline{b})|\!\} & \\
\text{Action}(P\backslash L, a) & = & \{\!|P'\backslash L | P' \in \text{Action}(P, a) \wedge a \notin L \wedge \overline{a} \notin L|\!\} & (7) \\
\text{Action}(P\backslash L, \tau) & = & \{\!|P'\backslash L | P' \in \text{Action}(P, \tau)|\!\} & (8) \\
\text{Action}(P[f], a) & = & \{\!|P'[f] | \exists b \in \Lambda. f(b) = a \wedge P' \in \text{Action}(P, b)|\!\} & (9) \\
\text{Action}(P[f], \tau) & = & \{\!|P'[f] | P' \in \text{Action}(P, \tau)|\!\} & (10) \\
\text{Action}(\mu X.P, \alpha) & = & \text{Action}(P\{\mu X.P/X\}, \alpha) & (11)
\end{array}
$$

Now we are in a position to define the transition system of Eager Timed CCS:

Definition 5.2.7 *The transition system of Eager Timed CCS is given by:*

$$\stackrel{\alpha}{\Longrightarrow}_u = \{\!|(P, P') | (u, \alpha) \in \Delta(P) \wedge P' \in \text{Action}(\text{Delay}(P, u), \alpha)|\!\}$$

It is clear that, by definition, Eager Timed CCS satisfies the maximal progress property (Theorem 5.2.8).

Theorem 5.2.8 *(Maximal Progress)*

If $P \stackrel{\alpha}{\Longrightarrow}_u P'$ for some $P' \in \mathcal{P}$, $u \in \mathcal{T}$ and $\alpha \in Act$, there is no $P'' \in \mathcal{P}$ and $\beta \in Act$ such that $P \stackrel{\beta}{\Longrightarrow}_v P''$ and $v \neq u$.

The following examples illustrate the behaviour of Eager Timed CCS agents.

Example 5.2.9 *The agent $a_3.nil$ has only a single transition:*

$$a_3.nil \xrightarrow{a}_3 nil$$

Example 5.2.10 *The agent $a_3.nil + \overline{a}_4.nil$ has a single transition:*

$$a_3.nil + \overline{a}_4.nil \xrightarrow{a}_3 nil$$

Eagerness prevents the transition $\xrightarrow{\overline{a}}_4 nil$.

Example 5.2.11 *The process $a_4.nil + \overline{a}_4.nil$ has two derivatives:*

$$a_4.nil + \overline{a}_4.nil \xrightarrow{a}_4 nil$$

$$a_4.nil + \overline{a}_4.nil \xrightarrow{\overline{a}}_4 nil$$

In this case, both transitions are possible because their lower bounds are equal.

Example 5.2.12 *The parallel composition $a_3.nil|\overline{a}_4.nil$ has only a single derivative:*

$$a_3.nil|\overline{a}_4.nil \xrightarrow{a}_3 nil|\overline{a}_1.nil \xrightarrow{\overline{a}}_1 nil|nil$$

Example 5.2.13 *The parallel composition $a_4.nil|\overline{a}_4.nil$, unlike Example 5.2.12, has three derivatives:*

$a_4.nil|\overline{a}_4.nil$
$\stackrel{a}{\Longrightarrow}_4 \; nil|\overline{a}_0.nil$
$\stackrel{\overline{a}}{\Longrightarrow}_0 \; nil|nil$

$a_4.nil|\overline{a}_4.nil$
$\stackrel{\overline{a}}{\Longrightarrow}_4 \; a_0.nil|nil$
$\stackrel{a}{\Longrightarrow}_0 \; nil|nil$

$a_4.nil|\overline{a}_4.nil$
$\stackrel{\tau}{\Longrightarrow}_4 \; nil|nil$

The additional transition $\stackrel{\tau}{\Longrightarrow}_4 nil|nil$ *is possible because the complementary* a *and* $\overline{a}$ *actions become enabled at the same time, and eagerness therefore does not preclude their interaction to form a* τ *action.*

Example 5.2.14 *Restriction can be used to force interaction between agents in a parallel composition, even when eagerness would otherwise prevent it:*

$(a_3.nil|\overline{a}_4.nil)\backslash\{a\} \stackrel{\tau}{\Longrightarrow}_4 \; (nil|nil)\backslash\{a\}$

Here the action $\stackrel{a}{\Longrightarrow}_3$ *is prevented by the restriction.*

Example 5.2.15 *Non-deterministic behaviour can arise in the absence of explicit choice operators, through multiple instances of the same action label in a parallel composition.*

$(a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}$
$\stackrel{\tau}{\Longrightarrow}_4 \; (nil|a_0.b_4.nil|nil)\backslash\{a\}$

$(a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}$
$\stackrel{\tau}{\Longrightarrow}_4 \; (a_0.nil|b_4.nil|nil)\backslash\{a\}$
$\stackrel{b}{\Longrightarrow}_4 \; (a_0.nil|nil|nil)\backslash\{a\}$

Example 5.2.16 *This example shows eagerness forcing choice.*

$(a_3.nil|a_5.b_4.nil|\overline{a}_4.nil)\backslash\{a\}$
$\stackrel{\tau}{\Longrightarrow}_4 \quad (nil|a_1.b_4.nil|nil)\backslash\{a\}$

The transition $\stackrel{\tau}{\Longrightarrow}_5$ *is not possible.*

5.2.1 Relating Eager Timed CCS to Timed CCS

Eager Timed CCS introduces eagerness in a number of places. Clauses (3) and (7) of Definition 5.2.2 respectively force explicit and implicit τ actions to occur as soon as possible. The definition of $\mathcal{I}_\tau$ forces complementary actions to perform an implicit τ as soon as both actions are enabled. Finally, Δ (Definition 5.2.4) only allows the earliest possible actions to be performed.

Eager Timed CCS permits a subset of the transitions allowed by Timed CCS, in the sense that $P \stackrel{\alpha}{\Longrightarrow}_u P'$ implies $P \stackrel{\alpha}{\longrightarrow}_u P'$, while the reverse implication does not hold.

Consider the agent $a_5.nil$. Timed CCS allows this agent to evolve by any transition $\stackrel{a}{\longrightarrow}_u nil$ with $u \geq 5$, while Eager Timed CCS permits only a single transition $\stackrel{a}{\Longrightarrow}_5 nil$. Similarly, $(\overline{a}_5.nil|a_8.nil)\backslash\{a\}$ can perform $\stackrel{\tau}{\longrightarrow}_u (nil|nil)\backslash\{a\}$ for any $u \geq 8$ but can only perform $\stackrel{\tau}{\Longrightarrow}_u (nil|nil)\backslash\{a\}$ for $u = 8$.

This suggests that:

$$\stackrel{\alpha}{\Longrightarrow}_u \subseteq \{(P,P') | (P,P') \in \stackrel{\alpha}{\longrightarrow}_u \wedge \neg\exists P''.P \stackrel{\beta}{\longrightarrow}_{u'} P'' \wedge u' < u\}$$

We can easily show that $\stackrel{\alpha}{\Longrightarrow}_u$ can be an even more restricted subset of $\stackrel{\alpha}{\longrightarrow}_u$. Consider the agent $P \stackrel{\text{def}}{=} (\overline{a}_5.nil|a_8.nil)$. Timed CCS permits

$$P \stackrel{\overline{a}}{\longrightarrow}_{u_1} \stackrel{a}{\longrightarrow}_{u_2} (nil|nil)$$

for any non-negative u_1 and u_2 satisfying $u_1 \geq 5 \wedge u_1 + u_2 \geq 8$. Timed CCS also permits

$$P \xrightarrow{a}_{u_1} \xrightarrow{\overline{a}}_{u_2} (nil|nil)$$

for any non-negative u_1 and u_2 satisfying $u_1 \geq 8$. Finally Timed CCS also allows

$$P \xrightarrow{\tau}_{u} (nil|nil)$$

for any $u \geq 8$, while in contrast, Eager Timed CCS permits precisely one execution:

$$P \stackrel{\overline{a}}{\Longrightarrow}_5 \stackrel{a}{\Longrightarrow}_3 (nil|nil)$$

Since eager actions can pre-empt other actions, Eager Timed CCS precludes certain sequences of actions which are permitted by Timed CCS.

Timed CCS models timeout through the upper bound on actions. Eagerness allows us to model certain forms of timeout in Eager Timed CCS, despite the removal of upper bounds. The agent $a_l^u.P$ can perform $\xrightarrow{a}_t$ for any $t \in \{l \ldots u\}$; if it does not perform that transition by time u then the action times out and the agent evolves into a state from which it can delay infinitely, but can perform no further actions. We can add a timeout behaviour to this process. $a_l^u.P + \tau_u^u.Q$ will only be capable of $\xrightarrow{\tau}_0 Q$ at time u. If this agent is placed in a parallel composition with another agent R and the action a restricted, then the new combined agent $((a_l^u.P + \tau_u^u.Q)|R)\backslash\{a\}$ can evolve $\xrightarrow{\tau}_t (P|R')\backslash\{a\}$ for some $t \in \{l \ldots u\}$ only if R can offer an action $\overline{a}$ during that period. Otherwise, the agent will evolve $\xrightarrow{\tau}_u (Q|R')\backslash\{a\}$.

In Eager Timed CCS, the agent $a_l.P + \tau_u.Q$ can evolve through $\stackrel{a}{\Longrightarrow}_l P$ if $l \leq u$ and through $\stackrel{\tau}{\Longrightarrow}_u P$ if $u \leq l$. The agent $((a_l.P + \tau_u.Q)|R)\backslash\{a\}$ can perform $\stackrel{\tau}{\Longrightarrow}_t (P|R')\backslash\{a\}$ for some $t \in \{l \ldots u\}$ only if R can offer an action $\overline{a}$ during that period. Otherwise, the agent will evolve $\stackrel{\tau}{\Longrightarrow}_u (Q|R')\backslash\{a\}$.

5.2.2 Equivalences

CCS and its derivatives equate processes on the basis of the actions which they can perform. A bisimulation is a relation between the state spaces of two agents in which each related pair of states can perform "the same" actions, and all pairs of derivatives reached by the two agents performing the same action also lie in the relation. Bisimulations of various strictnesses are obtained from different views of the meaning of "the same".

In the context of Eager Timed CCS, there are two broad classes of bisimulation equivalence: a timed equivalence which requires agents to perform actions at the same time, and time-abstracted equivalences which are not concerned with the time at which actions are performed.

Timed Equivalence

Definition 5.2.17 *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a timed bisimulation if for any agent $P, Q \in \mathcal{P}$, $P\mathcal{R}Q$ if and only if:*

$\forall \alpha \in Act$ *and* $\forall u \in \mathcal{T}$

1. *if* $P \stackrel{\alpha}{\Longrightarrow}_u P'$ *then* $\exists Q' \in \mathcal{P}. Q \stackrel{\alpha}{\Longrightarrow}_u Q'$ *and* $P'\mathcal{R}Q'$.

2. *if* $Q \stackrel{\alpha}{\Longrightarrow}_u Q'$ *then* $\exists P' \in \mathcal{P}. P \stackrel{\alpha}{\Longrightarrow}_u P'$ *and* $P'\mathcal{R}Q'$.

Definition 5.2.18 $\sim = \bigcup\{\mathcal{R} | \mathcal{R}$ *is a timed bisimulation* $\}$

Unfortunately, timed bisimulation is not a congruence; it is not preserved by the restriction operator. Consider the agents

$$P \stackrel{\text{def}}{=} a_2.nil + b_3.nil$$
$$Q \stackrel{\text{def}}{=} a_2.nil$$

Clearly $P \sim Q$, but $P\backslash\{a\} \not\sim Q\backslash\{a\}$ since $P\backslash\{a\}$ can evolve through the transition $\stackrel{b}{\Longrightarrow}_3 nil$ and $Q\backslash\{a\}$ cannot. It is usual for an appropriately defined strong bisimulation to be a congruence in calculi derived from CCS. The anomaly in this case is due to

the interaction of eagerness and restriction in Eager Timed CCS. Eagerness prevents $P \stackrel{b}{\Longrightarrow}_3 nil$, thereby allowing $P \sim Q$ to hold. Restriction prevents $P\backslash\{a\} \stackrel{a}{\Longrightarrow}_2 nil$, so $P\backslash\{a\} \stackrel{b}{\Longrightarrow}_3 nil$ is now the earliest possible action. Note, however, that Q can replace P in a context which is not within a restriction. For example $P|R \sim Q|R$, but in general $(P|R)\backslash L \not\sim (Q|R)\backslash L$.

Time-Abstracted Equivalences

We may well wish to demonstrate that two processes have equivalent behaviour in terms of the sequences of actions which they can perform, without regard to the time which passes between actions. This gives rise to the notion of time-abstracted bisimulation.

Definition 5.2.19 *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a strong time-abstracted bisimulation if for any agent $P, Q \in \mathcal{P}$, $P\mathcal{R}Q$ if and only if:*

$\forall \alpha \in Act$ *and* $\forall u \in \mathcal{T}$

1. *if* $P \stackrel{\alpha}{\Longrightarrow}_u P'$ *then* $\exists Q' \in \mathcal{P}.Q \stackrel{\alpha}{\Longrightarrow}_v Q'$ *for some* $v \in \mathcal{T}$ *and* $P'\mathcal{R}Q'$.

2. *if* $Q \stackrel{\alpha}{\Longrightarrow}_u Q'$ *then* $\exists P' \in \mathcal{P}.P \stackrel{\alpha}{\Longrightarrow}_v P'$ *for some* $v \in \mathcal{T}$ *and* $P'\mathcal{R}Q'$.

Definition 5.2.20 $\sim_{\mathcal{T}} = \bigcup\{\mathcal{R} | \mathcal{R}$ *is a strong time-abstracted bisimulation* $\}$

The agents

$$P \stackrel{\text{def}}{=} a_2.b_3.nil + c_3.d_4.nil$$
$$Q \stackrel{\text{def}}{=} a_4.b_1.nil$$

satisfy $P \sim_{\mathcal{T}} Q$ but clearly $P \not\sim Q$. Strong time-abstract bisimulation is a weaker notion of equality than strong timed bisimulation in a temporal sense, but is equally demanding behaviourally.

A strong time-abstracted bisimulation requires that both agents are capable of exactly the same sequences of actions, but places no constraints on the times at which these actions occur. It will often be convenient to abstract away from internal τ actions,

which leads to a weak time-abstracted bisimulation in which we allow internal actions τ to be matched by zero or more τ actions.

We define a transition $P \stackrel{\epsilon}{\longmapsto}_u P'$ which means P idles for time u without any observable actions. We write $P \longmapsto_u P'$ in place of $P \stackrel{\epsilon}{\longmapsto}_u P'$ for convenience.

Definition 5.2.21 *$P \stackrel{\epsilon}{\longmapsto}_{u_1+\cdots+u_n} P'$ if $P \stackrel{\tau}{\Longrightarrow}_{u_1} \cdots \stackrel{\tau}{\Longrightarrow}_{u_n} P'$ for $u_1, \ldots, u_n \in \mathcal{T}$ and we allow $P \stackrel{\epsilon}{\longmapsto}_0 P$.*

We also define a transition $P \stackrel{\alpha}{\longmapsto}_u P'$ which means that action α is observed at time u. Before α is performed, we allow finitely many internal τ actions. We also allow finitely many τ actions at the same time as α, that is, after zero delay.

Definition 5.2.22 *$P \stackrel{\alpha}{\longmapsto}_{u+v} P'$ if $P \stackrel{\epsilon}{\longmapsto}_u \stackrel{\alpha}{\Longrightarrow}_v \stackrel{\epsilon}{\longmapsto}_0 P'$ for some $u, v \in \mathcal{T}$.*

We also make use of the operator $\hat{}$, defined as follows.

Definition 5.2.23 *$\hat{\alpha} \stackrel{\text{def}}{=}$ if $\alpha = \tau$ then ϵ else α.*

Definition 5.2.24 *A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a weak time-abstracted bisimulation if for any agent $P, Q \in \mathcal{P}$, $P \mathcal{R} Q$ if and only if:*

$\forall \alpha \in \Lambda$ *and* $\forall u \in \mathcal{T}$

1. *if $P \stackrel{\alpha}{\longmapsto}_u P'$ then $\exists Q' \in \mathcal{P}$ such that $Q \stackrel{\hat{\alpha}}{\longmapsto}_v Q'$ for some $u, v \in \mathcal{T}$ and $P' \mathcal{R} Q'$.*

2. *if $Q \stackrel{\alpha}{\longmapsto}_u Q'$ then $\exists P' \in \mathcal{P}$ such that $P \stackrel{\hat{\alpha}}{\longmapsto}_v P'$ for some $u, v \in \mathcal{T}$ and $P' \mathcal{R} Q'$.*

Definition 5.2.25 $\approx_{\mathcal{T}} = \bigcup \{\mathcal{R} | \mathcal{R}$ *is a weak time-abstracted bisimulation* $\}$

Consider the agents:

$$P \stackrel{\text{def}}{=} a_4.\overline{b}_6.nil$$

$$Q \stackrel{\text{def}}{=} (a_2.\overline{c}_0.nil | c_3.\overline{d}_1.e_2.\overline{b}_4.nil | d_2.\overline{e}_1.nil) \backslash \{c, d, e\}$$

$P \approx_{\mathcal{T}} Q$ but $P \not\approx Q$ and $P \not\sim_{\mathcal{T}} Q$.

5.3 Reasoning about Execution Time

We are interested in defining EXEC(P), the time taken for an agent P to terminate. We define successful termination to be reaching a state which cannot perform any action, but may delay indefinitely. The agent nil has these properties, so we might define termination to be reaching nil.

In fact, we have to extend this definition, since many agents are indistinguishable from nil without being strictly identical to it. Consider, for example, the agent $nil|nil$. Like nil, it cannot perform any action, and can delay infinitely. We shall use strong timed bisimulation $\sim$ as our equivalence. In fact, this choice is arbitrary, since any of the equivalences which we have discussed will only equate nil with agents which cannot perform any actions.

We say that a process P evolves into process Q in time t, written $P \overset{t}{\rightsquigarrow} Q$, if $P \overset{t}{\rightsquigarrow} Q$ can be derived by the following rules:

$$\rightsquigarrow_1 \frac{}{P \overset{0}{\rightsquigarrow} P}(P \sim nil) \qquad \rightsquigarrow_2 \frac{P \overset{\alpha}{\Longrightarrow}_{u_1} Q \quad Q \overset{u_2}{\rightsquigarrow} R}{P \overset{u_1+u_2}{\rightsquigarrow} R}$$

We define successful termination as reaching a state which is bisimilar to nil. Hence

$$\text{EXEC}(P) = \{u | P \overset{u}{\rightsquigarrow} Q\}.$$

By definition there is no $P \in \mathcal{P}$ such that $nil \overset{\alpha}{\Longrightarrow}_u P$ for any $\alpha \in Act$ and $u \in \mathcal{T}$, so we can test an agent P for bisimilarity to nil by checking that it has no derivatives.

EXEC(P) provides a multi-set of execution times for the agent P giving us no information about the relative frequency of these executions. There is an implicit assumption that every execution is equally likely, so that the agent

$$P \overset{\text{def}}{=} a_2.nil + b_2.(c_3.nil + d_3.e_4.nil)$$

has

$$\mathrm{EXEC}(P) = \{\!|2,5,9|\!\}$$

and the executions

$$P \stackrel{a}{\Longrightarrow}_2 nil$$
$$P \stackrel{b}{\Longrightarrow}_2 \stackrel{c}{\Longrightarrow}_3 nil$$
$$P \stackrel{b}{\Longrightarrow}_2 \stackrel{d}{\Longrightarrow}_3 \stackrel{e}{\Longrightarrow}_4 nil$$

are judged equally likely.

For this to be true we need $P \stackrel{b}{\Longrightarrow}_2$ to be twice as likely as $P \stackrel{a}{\Longrightarrow}_2$. A more natural local view would insist that these transitions are equally likely, and the probability of a particular execution is derived from the probability of its component transitions.

We can define

$$\mathrm{FAIREXEC}(P) = \{\!|(u,p)|(P,1) \stackrel{u}{\rightsquigarrow} (Q,p)|\!\}$$

where $(P,p) \stackrel{t}{\rightsquigarrow} (Q,p')$ is defined by the following rules:

$$\rightsquigarrow_1 \frac{}{(P,p) \stackrel{0}{\rightsquigarrow} (P,p)}(P \sim nil) \qquad \rightsquigarrow_2 \frac{P \stackrel{\alpha}{\Longrightarrow}_{u_1} Q \quad \left(Q, \frac{p}{|\Delta(P)|}\right) \stackrel{u_2}{\rightsquigarrow} (R,p')}{(P,p) \stackrel{u_1+u_2}{\rightsquigarrow} (R,p')}$$

We can easily combine the elements of $\mathrm{FAIREXEC}(P)$ to form a set rather than a multi-set:

$$\mathrm{PEXEC}(P) = \{(u, \textstyle\sum_{(u,p)\in \mathrm{FAIREXEC}(P)} p) | (u,p) \in \mathrm{FAIREXEC}(P)\}$$

The following examples show the execution times of Eager Timed CCS agents.

Example 5.3.1 *The agent $a_3.nil$ can evolve only through one transition:*

$$a_3.nil$$
$$\stackrel{a}{\Longrightarrow}_3 \quad nil$$

This gives

$$\mathrm{EXEC}(a_3.nil) = \{|3|\}$$

and

$$\mathrm{PEXEC}(a_3.nil) = \{(3, 100\%)\}$$

Example 5.3.2 *The agent $a_3.nil + \overline{a}_4.nil$ can only perform one transition:*

$$a_3.nil + \overline{a}_4.nil \xrightarrow{a}_3 nil$$

giving

$$\mathrm{EXEC}(a_3.nil + \overline{a}_4.nil) = \{|3|\}$$

and

$$\mathrm{PEXEC}(a_3.nil + \overline{a}_4.nil) = \{(3, 100\%)\}$$

Example 5.3.3 *The agent $a_4.nil + \overline{a}_4.nil$ has two possible executions:*

$$a_4.nil + \overline{a}_4.nil \xrightarrow{a}_4 nil$$

$$a_4.nil + \overline{a}_4.nil \xrightarrow{\overline{a}}_4 nil$$

So

$$\mathrm{EXEC}(a_4.nil + \overline{a}_4.nil) = \{|4, 4|\}$$

and

$$\mathrm{PEXEC}(a_4.nil + \overline{a}_4.nil) = \{(4, 100\%)\}$$

Hence PEXEC *cannot distinguish $b_4.nil$ and $a_4.nil + \overline{a}_4.nil$, since they have identical execution profiles.*

Example 5.3.4 *Eagerness forces* $a_3.nil|\overline{a}_4.nil$ *to perform the transition* $\stackrel{a}{\Longrightarrow}_3$*:*

$$\begin{aligned} & a_3.nil|\overline{a}_4.nil \\ \stackrel{a}{\Longrightarrow}_3 \quad & nil|\overline{a}_1.nil \\ \stackrel{\overline{a}}{\Longrightarrow}_1 \quad & nil|nil \end{aligned}$$

This gives

$$\mathrm{EXEC}(a_3.nil|\overline{a}_4.nil) = \{\!|4|\!\}$$

and

$$\mathrm{PEXEC}(a_3.nil|\overline{a}_4.nil) = \{(4, 100\%)\}$$

Example 5.3.5 *The agent* $a_4.nil|\overline{a}_4.nil$ *has three possible executions:*

$$\begin{aligned} & a_4.nil|\overline{a}_4.nil \\ \stackrel{a}{\Longrightarrow}_4 \quad & nil|\overline{a}_0.nil \\ \stackrel{\overline{a}}{\Longrightarrow}_0 \quad & nil|nil \end{aligned}$$

$$\begin{aligned} & a_4.nil|\overline{a}_4.nil \\ \stackrel{\overline{a}}{\Longrightarrow}_4 \quad & a_4.nil|nil \\ \stackrel{a}{\Longrightarrow}_0 \quad & nil|nil \end{aligned}$$

$$\begin{aligned} & a_4.nil|\overline{a}_4.nil \\ \stackrel{\tau}{\Longrightarrow}_4 \quad & nil|nil \end{aligned}$$

This means that

$$\mathrm{EXEC}(a_4.nil|\overline{a}_4.nil) = \{\!|4, 4, 4|\!\}$$

and

$$\mathrm{PEXEC}(a_4.nil|\overline{a}_4.nil) = \{(4, 100\%)\}$$

Note that, despite the different number of executions permitted by $a_4.nil|\overline{a}_4.nil$ *and* $a_3.nil|\overline{a}_4.nil$ *(Example 5.3.4),* PEXEC *gives the same profile for both agents.*

Example 5.3.6 *The agent* $(a_3.nil|\overline{a}_4.nil)\backslash\{a\}$ *has only one possible execution:*

$$(a_3.nil|\overline{a}_4.nil)\backslash\{a\} \stackrel{\tau}{\Longrightarrow}_4 \ (nil|nil)\backslash\{a\}$$

giving

$$\mathrm{EXEC}((a_3.nil|\overline{a}_4.nil)\backslash\{a\}) = \{|4|\}$$

and

$$\mathrm{PEXEC}((a_3.nil|\overline{a}_4.nil)\backslash\{a\}) = \{(4, 100\%)\}$$

Example 5.3.7 *The agent* $(a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}$ *has two possible executions:*

$$\begin{aligned}&(a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}\\ &\stackrel{\tau}{\Longrightarrow}_4 \ (nil|a_0.b_4.nil|nil)\backslash\{a\}\end{aligned}$$

$$\begin{aligned}&(a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}\\ &\stackrel{\tau}{\Longrightarrow}_4 \ (a_0.nil|b_4.nil|nil)\backslash\{a\}\\ &\stackrel{b}{\Longrightarrow}_4 \ (a_0.nil|nil|nil)\backslash\{a\}\end{aligned}$$

which give

$$\mathrm{EXEC}((a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}) = \{|4, 8|\}$$

and

$$\mathrm{PEXEC}((a_3.nil|a_3.b_4.nil|\overline{a}_4.nil)\backslash\{a\}) = \{(4, 50\%), (8, 50\%)\}$$

Even if $P \sim Q$, we cannot guarantee $\mathrm{EXEC}(P) = \mathrm{EXEC}(Q)$. Consider the agents:

$$\begin{aligned}P &\stackrel{\mathrm{def}}{=} a_0.\tau_2.nil + b_0.\tau_4.nil\\ Q &\stackrel{\mathrm{def}}{=} a_0.\tau_2.nil + a_0.\tau_2.nil + a_0.\tau_2.nil + b_0.\tau_4.nil\end{aligned}$$

Clearly, $P \sim Q$, but $\mathrm{EXEC}(P) = \{|2, 4|\}$ while $\mathrm{EXEC}(Q) = \{|2, 2, 2, 4|\}$. Strong timed bisimulation is too weak an inequality to guarantee the structural equivalence required to ensure $\mathrm{EXEC}(P) = \mathrm{EXEC}(Q)$.

PEPA [Hillston, 1994] encounters a similar problem — strong bisimilarity of two PEPA agents does not guarantee equivalence of the underlying Markov processes.

5.3.1 Integrating Reasoning About Behaviour and Performance

The designer of a concurrent system is interested in behaviour as well as performance; it is important that the system behaves as intended. A variety of bisimulation techniques can be used to show the equivalence of two Eager Timed CCS agents, allowing the designer to demonstrate that a system is consistent with a specification. One particular challenge in designing concurrent systems is to ensure that the system cannot deadlock, by reaching a state in which no actions are possible.

In the previous discussion we considered defining successful termination as reaching nil, a state which exhibits precisely this property. The agent

$$D \stackrel{\text{def}}{=} (\overline{a}_3.\overline{b}_4.nil|a_5.b_6.nil)\backslash\{a,b\}$$

evolves by $\stackrel{\tau}{\Longrightarrow}_5\stackrel{\tau}{\Longrightarrow}_6 nil|nil$ and $\text{EXEC}(D) = \{|11|\}$. Suppose a design error had been made, resulting in the agent $D' = (\overline{a}_3.\overline{a}_4.nil|a_5.b_6.nil)\backslash\{a,b\}$. D' evolves by $\stackrel{\tau}{\Longrightarrow}_5$ to the deadlocked state $(\overline{a}_4.nil|b_6.nil)\backslash\{a,b\}$, which is bisimilar to nil. Since EXEC cannot distinguish between this deadlocked state and successful termination, we have $\text{EXEC} = \{|5|\}$, which is clearly misleading.

Our approach is to define EXEC for deadlock-free processes, and to test for deadlock by substituting any live process for nil.

Definition 5.3.8 *A process is* live *iff*

1. $P\stackrel{\alpha}{\Longrightarrow}_u P'$ *for some* $\alpha \in Act$ *and* $u \in \mathcal{T}$
2. P' *is live.*

By using the agent $\mu X.\tau_1.X$ to denote successful termination, we ensure that "terminated" processes are *live* in the sense that they can continually perform actions and allow time to pass. If an agent of this form can reach a state bisimilar to nil, we know that this can only occur through a deadlock. In fact, we can use any agent $\mu X.\tau_u.X$ with some $u \in \mathcal{T}$ such that $u \neq 0 \wedge u \neq \infty$. It is inconvenient to use this definition of successful termination when reasoning about execution time, since bisimilarity with $\mu X.\tau_1.X$ is not easily tested in a local way.

If we define agents using a distinguished process variable *Terminated* to denote successful termination, we can substitute an appropriate value of *Terminated* depending on the analysis in which we are interested. Given an agent P in this form, we define

$$P_{behavioural} = P\{\mu X.\tau.X/Terminated\} \text{ and } P_{performance} = P\{nil/Terminated\}.$$

We search for deadlocks in $P_{behavioural}$ and revise P if necessary until no deadlocks arise. Then we can evaluate $\text{EXEC}(P_{performance})$ safe in the knowledge that only valid execution times will appear. This is analogous to proving termination and correctness in conventional program proving situations.

5.3.2 Relating Agents in Terms of Performance

We might wish to compare agents on the basis of their performance. We could view agent P as *faster than* agent Q iff

$$\forall u_P \in \text{EXEC}(P).\forall u_Q \in \text{EXEC}(Q).u_P < u_Q$$

However, this takes no account of the behavioural equivalence of P and Q, and we could not simply replace Q by P without regard to their behaviour. We could make use of the time-abstracted equivalences discussed in Section 5.2.2 to determine the behavioural equivalance of a substitute before investigating whether it is faster than the agent which it is replacing.

A bisimulation "performance equivalence" combining behavioural and performance equivalence is presented in [Gorrieri and Rocetti, 1993]. Each action has a duration,

and each process updates a local clock as it performs actions. When two processes interact, their clocks are synchronised, with one waiting for the other if necessary. Two processes are deemed to be equivalent if they are capable of the same actions in the same period of time, but this equivalence is not a congruence.

Moller and Tofts discuss a similar problem which arises when we try to define a *faster than* relation based on bisimulation [Moller and Tofts, 1990]. Clearly, we would want to consider the agent $a_1.nil$ to be faster than $a_3.nil$. We have already seen that we can model timeout in Eager Timed CCS: the agent $P + b_2.nil$ allows process P to evolve until time 2, when it will perform the action b and evolve to nil. If we replace P with $a_1.nil$ and $a_3.nil$ we obtain the behaviour of $a_1.nil$ and $b_2.nil$ respectively. For our faster-than relation to be substitutive (i.e. a precongruence) we would have to accept that $a_1.nil$ is faster than $b_2.nil$. Moller and Tofts contend that this is undesirable because the two terms are not behaviourally equivalent.

The same problem arises in Eager Timed CCS, and can only be resolved by removing eagerness, or removing the $+$ operator.

5.3.3 Automated Analysis of Execution Time

Evaluating EXEC for a given agent P involves generating the state space of the agent, then traversing sequences of transitions from P to agents bisimilar to nil, calculating the execution time t of each sequence as given by $\overset{t}{\rightsquigarrow}$. Performing this process by hand is tedious, error-prone and only practical for small systems.

We can implement Δ (Definition 5.2.4), Delay (Definition 5.2.5) and Action (Definition 5.2.6) directly.

As it stands, $\mathcal{I}$ involves quantification over Act or Λ, despite the fact that many agents will only be capable of much smaller sets of action. With a syntactic definition of the set of possible next actions of an agent we can redefine these rules in a form more amenable to efficient implementation.

Definition 5.3.9 *$\mathcal{A} : \mathcal{P} \to \mathbb{F}\,(Act)$ gives the set of actions which an agent may possibly perform as its next action. Note how we assume that an implicit τ action may arise from a parallel composition. In this sense, $\mathcal{A}(P)$ is an upper bound on the set of actions which P may perform.*

$$
\begin{aligned}
\mathcal{A}(nil) &= \emptyset && (1)\\
\mathcal{A}(a_e.P) &= \{a\} && (2)\\
\mathcal{A}(\tau_e.P) &= \{\tau\} && (3)\\
\mathcal{A}(P+Q) &= \mathcal{A}(P) \cup \mathcal{A}(Q) && (4)\\
\mathcal{A}(P|Q) &= \mathcal{A}(P) \cup \mathcal{A}(Q) \cup \{\tau\} && (5)\\
\mathcal{A}(P\backslash L) &= \mathcal{A}(P) - (L \cup \overline{L}) && (6)\\
\mathcal{A}(P[f]) &= \{f(l) : l \in \mathcal{A}(P)\} && (7)\\
\mathcal{A}(\mu X.P) &= \mathcal{A}(P\{\mu X.P/P\}) && (8)
\end{aligned}
$$

Clauses (7) and (10) of Definition 5.2.2 can be redefined as:

$$
\begin{aligned}
\mathcal{I}(P|Q,\tau) &= \mathcal{I}(E,\tau) && (7)\\
&\cup\ \mathcal{I}(F,\tau)\\
&\cup\ \{[u,u]|\ \exists b \in \mathcal{A}(E) \cup \mathcal{A}(F).\\
&\qquad \exists [u_1,u_2] \in \mathcal{I}(E,b).\\
&\qquad \exists [u_3,u_4] \in \mathcal{I}(F,\overline{b}).\\
&\qquad ([u_1,u_2] \cap [u_3,u_4] \neq \emptyset) \wedge u = \max(u_1,u_3)\}
\end{aligned}
$$

$$
\mathcal{I}(P[f],\alpha) = \{[u_1,u_2]|\exists \beta \in \mathcal{A}(P).f(\beta) = \alpha \wedge [u_1,u_2] \in \mathcal{I}(P,\beta)\} \quad (10)
$$

We can also improve the definition of δ:

$$
\delta(P) = \bigcup_{\alpha \in \mathcal{A}(P)} \{(\alpha,[u_1,u_2])|[u_1,u_2] \in \mathcal{I}(P,\alpha)\}
$$

and Clause (9) of Definition 5.2.6:

$$
\mathrm{Action}(P[f],\alpha) = \{P'[f]|\exists \beta \in \mathcal{A}(L).f(\beta) = \alpha \wedge P' \in Action(P,\beta)\} \quad (9)
$$

A tool based on these definitions has been implemented in Standard ML. The tool consists of a front end which parses input files, creating an internal representation of the agent to be analysed, and a back end which generates the state space of the agent and analyses the agent's execution time(s).

Although parsing agent descriptions and generating state spaces are components of other analysis tools for process calculi, such as the Concurrency Workbench [Cleaveland *et al.*, 1989] and the PEPA Workbench [Gilmore and Hillston, 1994], the tool described here does not exploit code from these tools. The Concurrency Workbench is geared to much more sophisticated behavioural reasoning than we require. The PEPA Workbench has a stochastic model of time and relies on another application to analyse the stochastic processes it produces from descriptions of PEPA agents. The lexer and parser in the front end make use of SML-Lex [Appel *et al.*, 1992] and SML-Yacc [Tarditi and Appel, 1991].

In the back end, a simple approach to generating the state space can be very inefficient. The state space of an Eager Timed CCS agent is often a graph rather than a tree; the same state can be reached by several different sequences of individual actions. A naive state space generation algorithm expands the state space as a tree, leading to a combinatorial explosion agents in with parallel compositions. To illustrate this, we consider the agent

$$P \stackrel{\text{def}}{=} \prod_{i \in \{1,\ldots,n\}} a_{i_t}.Q_i$$

Let us assume that no two a_i actions can combine to give rise to a τ action

$$\forall i, j \in \{1, \ldots, n\}.\overline{a_i} \neq a_j$$

and that no Q_i can perform an immediate action, so that

$$\forall i \in \{1, \ldots, n\}.\neg\exists \alpha \in Act.Q_i \stackrel{\alpha}{\Longrightarrow}_0$$

Given those constraints, which simplify our analysis, we can see that there are $n!$ different sequences of actions from P which reach $\prod_{i\in\{1,\ldots,n\}} nil$. Naively generating this part of the state space gives rise to n immediate derivatives of P, $(n-1)$ derivatives of each of those, and so on to a total of

$$\sum_{i\in\{0,\ldots,n\}} \frac{n!}{(n-i)!}$$

states. However, many of those states are structurally identical to each other. For example, if $P \stackrel{a_1}{\Longrightarrow}_t \stackrel{a_2}{\Longrightarrow}_0 R_1$ and $P \stackrel{a_2}{\Longrightarrow}_t \stackrel{a_1}{\Longrightarrow}_0 R_2$ then R_1 and R_2 will be represented as separate states in the state space, even though they are clearly identical. In fact, there are only

$$\sum_{i\in\{0,\ldots,n\}} \frac{n!}{(n-i)!i!} = 2^n$$

different states in the state space. By checking for the presence of a state in the state space before adding it, we can avoid re-generating its state space. This brings crucial performance improvements in the analysis of larger agents.

The tool uses a simple test of structural identity to determine the equivalence of two states. Rather than exhaustively comparing with each state in the state space, the tool maintains a hash table of states, to quickly identify candidate states for comparison. The hash table implementation draws on modules from the Standard ML of New Jersey library [AT&T Bell Laboratories, 1993].

The tool was used to analyse the following agents.

Example 5.3.10 *The results of analysing the input file:*

```
define P1 = (a,3).nil
analyse P1
```

are:

```
Execution times are:
  3
Summary:
  3      (100.0%)
```

Example 5.3.11 *The results of analysing the input file:*

```
define P2 = (a,3).nil + (a',4).nil
analyse P2
```

are:

```
Execution times are:
  3
Summary:
  3      (100.0%)
```

Example 5.3.12 *The results of analysing the input file:*

```
define P3 = (a,4).nil + (a',4).nil
analyse P3
```

are:

```
Execution times are:
  4
  4
Summary:
  4      (100.0%)
```

Example 5.3.13 *The results of analysing the input file:*

```
define P4 = (a,3).nil|(a',4).nil
analyse P4
```

are:

```
Execution times are:
  4
Summary:
  4      (100.0%)
```

Example 5.3.14 *The results of analysing the input file:*

```
define P5 = (a,4).nil|(a',4).nil
analyse P5
```

are:

```
Execution times are:
  4
  4
  4
Summary:
  4      (100.0%)
```

Example 5.3.15 *The results of analysing the input file:*

```
define P6 = ((a,3).nil|(a',4).nil)\{a}
analyse P6
```

are:

```
Execution times are:
  4
Summary:
  4      (100.0%)
```

Example 5.3.16 *The results of analysing the input file:*

```
define P7 = ((a,3).nil|(a,3).(b,4).nil|(a',4).nil)\{a}
analyse P7
```

are:

```
Execution times are:
  4
  8
Summary:
  4       (50.0%)
  8       (50.0%)
```

None of the above analyses took significant time on a SPARCstation-20 configured with 96 Mbyte of RAM. The `time` command reported elapsed times of between 0.0s and 0.1s for each analysis. However, the challenge is to achieve efficient analysis of larger, more complex models. As it stands the tool's scalability is limited by it's approach to state space generation, which involves redundant computation and, more importantly, makes inefficient use of memory. There is scope to use semantic equivalences rather than structural identity to reduce the size of the state space. For example, we can exploit symmetry in models. Rather than allowing:

$$(a_0.nil|a_0.nil|a_0.nil)$$

to have three derivatives:

$$\begin{aligned} &\stackrel{a}{\Longrightarrow}_0 \ (nil|a_0.nil|a_0.nil) \\ &\stackrel{a}{\Longrightarrow}_0 \ (a_0.nil|nil|a_0.nil) \\ &\stackrel{a}{\Longrightarrow}_0 \ (nil|a_0.nil|a_0.nil) \end{aligned}$$

we could notice that:

$$\begin{aligned} &(nil|a_0.nil|a_0.nil) \\ &\sim \ (a_0.nil|nil|a_0.nil) \\ &\sim \ (a_0.nil|a_0.nil|nil) \end{aligned}$$

and exploit this symmetry by allowing a single transition with an extra label:

$$(a_0.nil|a_0.nil|a_0.nil)$$
$$(3)\stackrel{a}{\Longrightarrow}_0 \quad (nil|a_0.nil|a_0.nil)$$

where $(n)\stackrel{\alpha}{\Longrightarrow}_u P$ means that there are n transitions to agents which are bisimilar to P. It is important to maintain information about the multiplicity of derivatives so that analysis of this improved representation of state space produces the same frequencies for each derivation sequence and therefore the same probabilities for each execution time.

To implement such a technique fully, the derivative agent of every second and later transition identified from a state would have to be tested for equivalence with derivatives which have already been found. The cost of this testing is likely to be significant. Alternatively, a structural test, which equates permutations of parallel compositions within agents, might be more tractable and still result in important savings.

5.4 Related Work

Pure process algebras abstract away from time, providing instantaneous actions and representing the ordering of actions only. There have been many proposals for extending process algebras with notions of time. This work typically focuses on one of two objectives: reasoning about behavioural properties of timed systems or reasoning about the performance of timed systems.

5.4.1 Reasoning About Behaviour

The simplest way of introducing time into a process calculus is to define a synchronous calculus, in which processes evolve in synchrony with a global clock, each performing an action at every tick. An example is SCCS [Milner, 1983].

Alternatively, we can allow agents to delay for specified periods. Temporal CCS [Moller and Tofts, 1990] provides timed delay transitions labelled with natural numbers as well as instantaneous action transitions. Automated analysis of Temporal CCS agents is provided by the Concurrency Workbench [Cleaveland *et al.*, 1993]. Timed CCS [Chen, 1993] introduces upper and lower bounds on the time at which actions are possible. It also caters for explicit time variables, allowing a greater variety of time-dependent behaviours to be described.

Real time can be introduced to CSP through a delay mechanism, either as a separate operator or as part of the prefix operator [Reed and Roscoe, 1986; Davies and Schneider, 1989].

In Real Time ACP [Baeten and Bergstra, 1991], actions can be associated with an absolute time, a relative time, or an interval.

An algebra of timed processes with real-valued clocks is presented in [Yi *et al.*, 1994]. Actions are predicated on constraints over a set of clocks and reachability analyses are performed using constraint solving techniques.

ACSR [Brémond-Grégoire, 1994; Lee *et al.*, 1994] distinguishes between timed actions that consume resources and instantaneous actions used for synchronisations. ACSR has static priorities, is eager, and provides a strong bisimulation equivalence. It is primarily used for reasoning about real time constraints on behaviour, but has also been applied to analysing the performance of superscalar processors [Choi *et al.*, 1994]. A tool has been developed to analyse ACSR models [Clarke *et al.*, 1995].

5.4.2 Reasoning About Performance

One attraction of integrating performance modelling with process algebra is the potential for using bisimulation equivalences to simplify and structure models at a high level, rather than relying on techniques to decompose and simplify the underlying numerical model.

PEPA [Hillston, 1994] models the duration of actions with an exponentially distributed random variable. PEPA's cooperation combinator defines a set of "shared" actions that will only be enabled when the combined processes are ready to perform them. An operational semantics for PEPA can be used to generate a continuous time Markov process from any PEPA model. Various bisimulation equivalences are defined and used to simplify analysis at the model level, before numerical models are generated. An analysis tool solves the underlying Markov process numerically [Gilmore and Hillston, 1994].

TIPP [Hermanns *et al.*, 1994] also models the duration of actions as exponentially distributed random variables. Extensions to the language cater for other distributions.

Simulation modellers have also made use of timed process algebras, in an effort to prove behavioural properties of simulation models.

CCS+, defined in [Strulo, 1993], provides time evolution and probability evolution as well as actions, and is intended for reasoning about simulations. Bisimulation equivalences can be used to simplify models.

Extended Activity Diagrams, a graphical representation of simulation models, are presented in [Pooley, 1995a; Pooley, 1995b]. Translations to CCS and Temporal CCS are provided to facilitate behavioural reasoning, while a translation to DEMOS allows simulation experiments to be conducted.

5.5 Conclusions

We have presented an eager extension of Timed CCS, and shown how execution time metrics can be derived from models. An automated tool implementing these analyses has been demonstrated and shown to be efficient for small models. We have also discussed possible enhancements of the tool to improve its efficiency in analysing larger, more complex models. In the next chapter, we use Eager Timed CCS to model parallel applications.

Chapter 6

Modelling Parallel Applications in Eager Timed CCS

In Chapter 5 we defined Eager Timed CCS and showed how measures of execution time can be derived from Eager Timed CCS agents. In this chapter we apply these techniques to formal models of parallel applications.

We introduce some convenient syntactic sugar before considering how we can model processes and processors in Eager Timed CCS. We then extend this basic model to describe multi-tasking and multi-processing systems. The critical component of parallel applications is inter-process communication, and we show how models can be extended to capture typical message-passing operations, and resource contention in interconnection networks. Using these techniques, we construct two models of a parallel application, and analyse their performance.

6.1 Preliminaries

Milner [Milner, 1989] defines a value-passing calculus in which agents can communicate values through interactions. The central idea is that each label in the value-passing calculus corresponds to a set of labels in the basic calculus, each associated with a fixed value.

The set of agent expressions in value-passing Eager Timed CCS is defined by

$$\begin{aligned} E \;::=\; & a(x)@t_u.E \mid \overline{a}(e)@t_u.E \mid \tau@t_u.E \mid E+F \mid E|F \mid E\backslash L \mid E[f] \\ \mid\; & X \mid A(e_1,\ldots,e_n) \end{aligned}$$

We assume value expressions e built up from value variables $x, y, \ldots$, value constants v and any operators we require.

The prefix notation $a.P$ is augmented with an argument to give $a(x).P$ which is interpreted as receiving a value through an action a, with the received value becoming bound to x in P. The agent $\overline{a}(v).P$ is interpreted as sending a value v to another process through an action $\overline{a}$, then evolving to P. We extend Eager Timed CCS similarly, and define the semantics of the message-passing calculus by translation to Eager Timed CCS.

Definition 6.1.1 *The translation $[\![\ ]\!] : \mathcal{P}_{vp} \rightarrow \mathcal{P}$ from value-passing agents $\mathcal{P}_{vp}$ to $\mathcal{P}$ is defined as follows, given a set V of possible values:*

$$\begin{aligned}
[\![X]\!] &= X && (1)\\
[\![a(x)@t_u.P]\!] &= \textstyle\sum_{v\in V} a^v@t_u.[\![P\{v/x\}]\!] && (2)\\
[\![\overline{a}(v)@t_u.P]\!] &= \overline{a}^v@t_u.[\![P]\!] && (3)\\
[\![\tau@t_u.P]\!] &= \tau@t_u.[\![P]\!] && (4)\\
[\![P+Q]\!] &= [\![P]\!]+[\![Q]\!] && (5)\\
[\![P|Q]\!] &= [\![P]\!]|[\![Q]\!] && (6)\\
[\![P\backslash L]\!] &= [\![P]\!]\backslash\{l^v : l\in L, v\in V\} && (7)\\
[\![P[f]]\!] &= [\![P]\!][f'] \text{ where } f'(l^v) = f(l)^v && (8)\\
[\![A(e_1,\ldots,e_n)]\!] &= A_{e_1,\ldots,e_n} && (9)
\end{aligned}$$

Note that the translation to Eager Timed CCS considers only agent expressions which contain no free variables. If an expression contains a free variable, we would treat it as a family of expressions in which various values were substituted for the free variable.

6.2 Modelling Processes and Processors

We can view a process and a processor which is executing it as a concurrent system in Eager Timed CCS. A process has a certain behaviour, which is implemented by a processor, taking a certain amount of time. The behaviour of the process is independent of the processor on which it is executed, but it is only meaningful to talk about the performance of a process in the context of a particular processor.

By defining a set of operations $\mathcal{O} = \{o_1, \ldots, o_n\}$ which a processor can perform on behalf of a process, and a mapping $T : \mathcal{O} \rightarrow \mathcal{T}$ representing the execution time of each operation, we can provide a simple model of a processor and a process which captures this notion.

A process can be modelled by an agent which requests a sequence of operations:

$$\text{Process} \stackrel{\text{def}}{=} \overline{o_{10}}.\overline{o_{20}}.\ldots.\overline{o_{n0}}.nil$$

Notice that the process model makes no statement about time. It merely waits for its environment to be able to accept its next request for a service. It is, in this sense, a purely behavioural model in keeping with our goal of separating behavioural and temporal reasoning. Considered in isolation, the process executes in zero time.

To reason about the performance of our process, we must consider it in the context of a particular processor.

$$(\text{Processor}|\text{Process})\backslash\mathcal{O}$$

A simple processor can be modelled by an agent which repeatedly accepts a request for an operation and delays for a certain period, representing the time taken to perform the requested operation:

$$\text{Processor} \stackrel{\text{def}}{=} \sum_{o \in \mathcal{O}} o_0.\tau_{T(o)}.\text{Processor}$$

Example 6.2.1 *Suppose we have* $\mathcal{O} = \{\text{add}, \text{mult}\}$ *with* $T(\text{add}) = 3$ *and* $T(\text{mult}) = 4$ *then our simple processor would be:*

$$\begin{aligned} \text{Processor} \stackrel{\text{def}}{=} & \ \text{add}_0.\tau_3.\text{Processor} \\ + & \ \text{mult}_0.\tau_4.\text{Processor} \end{aligned}$$

Consider the processes:

$$\begin{aligned} \text{Process}_0 &\stackrel{\text{def}}{=} \overline{\text{add}}_0.\overline{\text{add}}_0.\overline{\text{mult}}_0.\overline{\text{mult}}_0.nil \\ \text{Process}_1 &\stackrel{\text{def}}{=} \overline{\text{add}}_0.\overline{\text{mult}}_0.\overline{\text{add}}_0.\overline{\text{mult}}_0.nil \end{aligned}$$

We have

$$\begin{aligned} \text{EXEC}((\text{Process}_0 | \text{Processor}) \backslash \mathcal{O}) &= \{\!|14|\!\} \\ \text{EXEC}((\text{Process}_1 | \text{Processor}) \backslash \mathcal{O}) &= \{\!|14|\!\} \end{aligned}$$

We can construct more complicated models of processors.

Example 6.2.2 *Consider a super-scalar processor which provides* $\mathcal{O} = \{\text{add}, \text{mult}\}$ *with* $T(\text{add}) = 3$ *and* $T(\text{mult}) = 4$ *as before, but has two functional units, so* add *and* mult *operations can be overlapped.*

$$\text{Controller} \stackrel{\text{def}}{=} \text{add}_0.\overline{\text{adder}}_0.\tau_1.\text{Controller}$$

$$
\begin{aligned}
&+ \quad \mathrm{mult}_0.\overline{\mathrm{multiplier}}_0.\tau_1.\mathrm{Controller} \\
\mathrm{Adder} &\stackrel{\mathrm{def}}{=} \mathrm{adder}_0.\tau_3.\mathrm{Adder} \\
\mathrm{Multiplier} &\stackrel{\mathrm{def}}{=} \mathrm{multiplier}_0.\tau_4.\mathrm{Multiplier} \\
\mathrm{SSProcessor} &\stackrel{\mathrm{def}}{=} (\mathrm{Controller}|\mathrm{Adder}|\mathrm{Multiplier})\backslash\{\mathrm{adder}, \mathrm{multiplier}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{EXEC}((\mathrm{Process}_0|\mathrm{SSProcessor})\backslash\mathcal{O}) &= \{|12|\} \\
\mathrm{EXEC}((\mathrm{Process}_1|\mathrm{SSProcessor})\backslash\mathcal{O}) &= \{|9|\}
\end{aligned}
$$

We can show that Processor $\approx_{\mathcal{T}}$ SSProcessor by constructing an appropriate bisimulation.

6.2.1 Multi-tasking

Two or more independent processes being executed by a single processor can be readily modelled:

$$(\mathrm{Process}_0|\mathrm{Process}_1|\ldots|\mathrm{Process}_{n-1}|\mathrm{Processor})\backslash\mathcal{O}$$

Example 6.2.3 *Suppose that* $\mathcal{O} = \{\mathrm{add}, \mathrm{mult}\}$ *with* $T(\mathrm{add}) = 3$ *and* $T(\mathrm{mult}) = 4$.

Given

$$
\begin{aligned}
\mathrm{Process}_0 &\stackrel{\mathrm{def}}{=} \overline{\mathrm{add}}_0.\overline{\mathrm{add}}_0.\overline{\mathrm{mult}}_0.\overline{\mathrm{mult}}_0.nil \\
\mathrm{Process}_1 &\stackrel{\mathrm{def}}{=} \overline{\mathrm{add}}_0.\overline{\mathrm{mult}}_0.\overline{\mathrm{add}}_0.\overline{\mathrm{mult}}_0.nil \\
\mathrm{Processor} &\stackrel{\mathrm{def}}{=} \mathrm{add}_0.\tau_3.\mathrm{Processor} \\
&+ \quad \mathrm{mult}_0.\tau_4.\mathrm{Processor}
\end{aligned}
$$

we have

$$\mathrm{PEXEC}((\mathrm{Process}_0|\mathrm{Process}_1|\mathrm{Processor})\backslash\{\mathrm{add}, \mathrm{mult}\}) = \{(28, 100\%)\}$$

In practice, there is a cost associated with a processor context-switching between processes — so processors tend to perform a series of operations on behalf of a particular process before servicing another process.

Example 6.2.4 *Let* $\mathcal{O}$, $\mathcal{T}$, Process_0 *and* Process_1 *be as in Example 6.2.3 above, with* o *used to range over* $\mathcal{O}$.

Our processor can either be dedicated to servicing a particular process, or looking for a process in need of service. We model the first state as $\text{Processing}(i,p)$ *where process* i *of* p *is being serviced.*

$$\begin{aligned}\text{Processing}(i,p) \stackrel{\text{def}}{=} & \sum_{o\in\mathcal{O}} o_{i0}.\tau_{T(o)}.\text{Processing}(i,p) \\ + & \ \tau_1.\text{Processor}(p)\end{aligned}$$

The τ *action allows the processor to time out while waiting for the process which it is servicing to request a further operation. This prevents the processor waiting for further requests after the process has completed.*

The uncommitted processor is modelled as:

$$\text{Processor}(p) \stackrel{\text{def}}{=} \sum_{i=0}^{p-1}\sum_{o\in\mathcal{O}} o_{i0}.\tau_c.\tau_{T(o)}.\text{Processing}(i,p)$$

where c *is the context switch time, which we will set to* 5.

Given the functions $f_i : \Lambda \to \Lambda$ *which map* $o \in \mathcal{O}$ *to* o_i, *we can analyse*

$$\text{System} \stackrel{\text{def}}{=} (\text{Processing}(0,2)|\text{Process}_0[f_0]|\text{Process}_1[f_1])\backslash\bigcup_{i=0}^{1}\bigcup_{o\in O}\{o_i\}$$

and obtain

$$\text{PEXEC}(\text{System}) = \{(35, 100\%)\}$$

In System *the execution of the two processes is serialised; once the processor begins servicing a process, it continues to do so until the process completes. We can extend our processor model with a notion of time quanta, using the agent:*

$$\text{Timer} \stackrel{\text{def}}{=} \overline{\text{time}_q}.nil + \text{cancel}_0.nil$$

where q is the maximum period for which the processor will be dedicated to a single processor. For the current example, we will set $q = 10$.

We modify the uncommitted processor to start a timer when it begins to service a particular process:

$$\begin{aligned}\text{Processor}(p) \stackrel{\text{def}}{=} & \sum_{i=0}^{p-1}\sum_{o\in\mathcal{O}} o_{i0}.\tau_c.(((\tau_{T(o)}.\text{Processing}(i,p))|\text{Timer}) \\ & \setminus \{\text{time}, \text{cancel}\})\end{aligned}$$

The timer must be able to interrupt the dedicated processor. Equally, the dedicated processor must be able to cancel the timer if the process completes.

$$\begin{aligned}\text{Processing}(i,p) \stackrel{\text{def}}{=} & \sum_{o\in\mathcal{O}} o_{i0}.\tau_{T(o)}.\text{Processing}(i,p) \\ & + \tau_1.\overline{\text{cancel}_0}.\text{Processor}(p) \\ & + \text{time}_0.\text{Processor}(p) \\ \text{System} \stackrel{\text{def}}{=} & (\text{Processing}(0,2)|\text{Process}_0[f_0]|\text{Process}_1[f_1]) \\ & \setminus \bigcup_{i=0}^{1}\bigcup_{o\in O}\{o_i\}\end{aligned}$$

Using the analysis tool, we obtain:

$$\text{PEXEC}(\text{System}) = \{(33, 25\%), (39, 50\%), (45, 25\%)\}$$

The range of execution times arise from non-determinism due to a race condition between the processor accepting the time out from the timer and accepting a further operation from the process. This race condition would give rise to an explosion of possibilities if the processes were longer.

This example still does not guarantee fairness in scheduling but, unlike the previous version, it does not preclude fair schedules.

6.3 Modelling Inter-process Communication

To model parallel applications, we must model inter-process communication. The simplest message-passing system allows processes to:

1. send messages to other processes;

2. receive messages

In order to describe message-passing operations, we must consider issues of addressing, synchronicity and blocking behaviour.

Addressing: We need an address scheme so that a process can define the destination to which a message is to be sent. In keeping with many message-passing systems, we will adopt a numerical addressing scheme in which processes are identified by a contiguous sequence of numerical addresses, starting from zero.

Synchronicity: A synchronous message-passing system guarantees that there will be a period of time in which corresponding send and receive operations will both be executing; in other words, that there will be a synchronisation between the sending and receiving processes. An asynchronous system makes no such guarantee. Synchronicity can be thought of as defining the point at which a sending process considers a communication to have been completed: after the message

is received (in a synchronous system); or after the message has been accepted by the message-passing system (in an asynchronous system). A receiving process considers both synchronous and asynchronous communications to have been completed when the message is received. Synchronicity is a property of the communication; sender and receiver must agree on the synchronicity of a message.

Blocking behaviour: A blocking communication operation cannot complete until the communication in which it is involved completes. Hence a blocking synchronous send operation cannot complete until the message is received at the the destination process. A non-blocking operation can complete before its communication completes, and is typically used in conjunction with a further operation which allows a process to test whether its outstanding non-blocking communication operations have completed. The importance of non-blocking operations lies in their support for overlapping communications with other communications or with computation. A blocking communication is, in fact, a special case of non-blocking in which no other operations are carried out between the initiation of the communication and the test for its completion. Blocking is a local property; a send can be blocking or non-blocking irrespective of whether the corresponding receive is blocking or non-blocking.

We will assume a static set of processes and neglect dynamic process management, This is in the spirit of the current MPI definition, and is representative of the majority of parallel applications. We will also assume a single process per processor, so our agents will take the form

$$((\text{Process}_0|\text{Processor})\backslash L|\ldots|(\text{Process}_{n-1}|\text{Processor})\backslash L)$$

We could allow the Eager Timed CCS representations of processes to communicate directly. A process would send to process i by performing a $\overline{\text{message}}_i$ action, while process i would receive a message by performing a message_i action. While attractively simple, this model is unsatisfactory for several reasons: it neglects the computational overhead

involved in sending and receiving a message; it neglects the latency of message transit; and it cannot easily model non-blocking or asynchronous message-passing. Instead, we will extend the processor model to support inter-process communication.

We do this by introducing four additional services which a process can request: send, sent, receive and received. The send and sent services are subscripted with the address of the destination process. The send_i service initiates the sending of a message to process with identifier i. A subsequent sent_i operation blocks until the message has actually been sent. The receive service asks the processor to be ready to receive a message, while the subsequent received service blocks until a message is in fact received.

$$\begin{aligned} \text{Processor} &\stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{send}_{i_0}.\tau_{t_{\text{send}}}.(\text{Processor}|\text{Sending}(i)) \\ &+ \text{receive}_0.\tau_{t_{\text{receive}}}.(\text{Processor}|\text{Receiving}) \end{aligned}$$

$$\text{Sending}(i) \stackrel{\text{def}}{=} \overline{\text{transmit}_{i0}}.\text{ack}_0.\text{sent}_0.nil$$

$$\text{Receiving} \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{transmit}_{i0}.\overline{\text{ack}_{i0}}.\text{received}_0.nil$$

Given models of Process_i for $(0 \leq i < p)$ where p is the number of processes, we model each node in our parallel system as:

$$\text{Node}_i \stackrel{\text{def}}{=} (\text{Process}_i|\text{Processor})\backslash L)$$

where

$$L = \bigcup_{i=0}^{p-1} \{\text{send}_i, \text{sent}_i\} \cup \{\text{receive}, \text{received}\}$$

We can define the whole system as:

$$\text{System} \stackrel{\text{def}}{=} \left(\prod_{i=0}^{n-1} \text{Node}_i\right) \setminus \bigcup_{i=0}^{n-1} \{\text{transmit}_i, \text{ack}_i\}$$

where

$$\prod_{i=a}^{b} P_i = P_a | P_{a+1} | \ldots | P_{b-1} | P_b$$

Example 6.3.1 *Let us consider a system of two processes defined as follows:*

$$\text{Process}_0 \stackrel{\text{def}}{=} \overline{\text{send}}_{10}.\overline{\text{sent}}_{10}.nil$$
$$\text{Process}_1 \stackrel{\text{def}}{=} \overline{\text{receive}}_0.\overline{\text{received}}_0.nil$$

To analyse these processes, or rather the agent constructed from them in the manner described above, we require values for t_{send} *and* t_{receive}. *Using* 100 *and* 50 *respectively results in* $\text{PEXEC}(\text{System}) = \{(100, 100\%)\}$.

Our model has no notion of network latency — messages are transmitted instantaneously from source to destination. We can introduce a network latency by redefining Sending and Receiving:

$$\text{Sending}(i) \stackrel{\text{def}}{=} \overline{\text{transmit}}_{i t_{\text{latency}}}.\text{ack}_0.\text{sent}_0.nil$$
$$\text{Receiving} \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{transmit}_{i0}.\overline{\text{ack}}_{i t_{\text{acklatency}}}.\text{received}_0.nil$$

In general, t_{latency} will depend upon the size of the message being sent. By requiring $\overline{\text{send}}$ actions to pass a message length value, we can parameterise Sending on the message length and include an expression for network latency.

$$\text{Processor} \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{send}_{i_0}(s).\tau_{t_{\text{send}}}.(\text{Processor} | \text{Sending}(i, s))$$

$$+ \quad \text{receive}_0.\tau_{t_{\text{receive}}}.(\text{Processor}|\text{Receiving})$$

$$\text{Sending}(i,s) \stackrel{\text{def}}{=} \overline{\text{transmit}}_{i(s\times b)}.\text{ack}_0.\text{sent}_0.nil$$

$$\text{Receiving} \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{transmit}_{i0}.\overline{\text{ack}}_{ib}.\text{received}_0.nil$$

Here we use a linear model for network latency, where b is the inverse of the bandwidth, i.e. the time taken to pass a single unit of data down the channel. We assume that the acknowledgement is a single unit of data.

Example 6.3.2 *Let's assume that we have two processes as in the previous example, and that the message size is 1000 units.*

$$\text{Process}_0 \stackrel{\text{def}}{=} \overline{\text{send}}_{10}(1000).\overline{\text{sent}}_{10}.nil$$
$$\text{Process}_1 \stackrel{\text{def}}{=} \overline{\text{receive}}_0.\overline{\text{received}}_0.nil$$

Again, we will set

$$t_{\text{send}} = 100$$

and

$$t_{\text{receive}} = 50$$

With $b = 1$, we obtain

$$\text{PEXEC(System)} = \{(1101, 100\%)\}$$

Our model does not include any scope for network contention. We can introduce this possibility by redefining Sending and Receiving and introducing a model of the network resource.

$$\text{Sending}(i,s) \stackrel{\text{def}}{=} \overline{\text{get}_0}.\overline{\text{release}_{(s \times b)}}.\overline{\text{transmit}_{i0}}.\text{ack}_0.\text{sent}_0.nil$$

$$\text{Receiving} \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} \text{transmit}_{i0}.\overline{\text{get}_0}.\overline{\text{release}_b}.\overline{\text{ack}_{i0}}.\text{received}_0.nil$$

$$\text{Network} \stackrel{\text{def}}{=} \text{get}_0.\text{release}_0.\text{Network}$$

The agent Network is essentially a semaphore controlling access to the network. Sending and Receiving acquire this semaphore then delay for periods corresponding to the message transfer time before releasing it. This ensures serialisation of all message transfers, as would occur on a bus. We have to modify our notion of a system to include the network agent:

$$\text{System} \stackrel{\text{def}}{=} \left(\left(\prod_{i=0}^{n-1} \text{Node}_i \right) | \text{Network} \right) \backslash \{\text{get}, \text{release}\} \cup \bigcup_{i=0}^{n-1} \{\text{transmit}_i, \text{ack}_i\}$$

Example 6.3.3 *Repeating Example 6.3.2 with our new definitions gives*

$$\text{PEXEC}(\text{System}) = \{(1101, 100\%)\}$$

as before — there is only a single message and a single acknowledgement, and therefore no potential for contention.

By replacing Network with another agent which ensures that no contention arises, we can model contention-free execution in the same framework and isolate the changes to our models in the choice of network agent.

$$\begin{aligned} \text{NetworkPerfect} \stackrel{\text{def}}{=} & \ \text{get}_0.\text{NetworkPerfect} \\ + & \ \text{release}_0.\text{NetworkPerfect} \end{aligned}$$

By adding more state to the model we could add a number of network resources, such as different links in an interconnection topology, and model contention at finer resolution.

6.3.1 Scatter

Now that we have seen how point-to-point communication can be modelled, we can consider collective operations built from point-to-point communications, such as scatter.

The simplest implementation of scatter is a repeated send.

$$\begin{aligned} \text{Process}_0 &\stackrel{\text{def}}{=} \overline{\text{send}_{1_0}}(s).\overline{\text{sent}_{1_0}}.\ldots.\overline{\text{send}_{p-1_0}}(s).\overline{\text{sent}_{p-1_0}}.nil \\ \text{Process}_i &\stackrel{\text{def}}{=} \overline{\text{receive}_0}.\overline{\text{received}_0}.nil \ \text{(for } 1 \leq i < p\text{)} \end{aligned}$$

A potentially more efficient implementation of scatter does not wait for each message to be delivered before sending the next.

$$\begin{aligned} \text{Process}_0 &\stackrel{\text{def}}{=} \overline{\text{send}_{1_0}}(s).\ldots.\overline{\text{send}_{p-1_0}}(s).\overline{\text{sent}_{1_0}}.\ldots.\overline{\text{sent}_{p-1_0}}.nil \\ \text{Process}_i &\stackrel{\text{def}}{=} \overline{\text{receive}_0}.\overline{\text{received}_0}.nil \ \text{(for } 1 \leq i < p\text{)} \end{aligned}$$

Table 6–1 presents the execution times of these two implementations on a bus network, obtained by analysis of the Eager Timed CCS descriptions. The parameter values used are: $t_{\text{send}} = 100$, $t_{\text{receive}} = 100$, $b = 1$, and $s = 1000$. The analysis tool's memory requirements exceeded the available memory on the system for the non-blocking model on 7 or more nodes.

The non-blocking implementation performs slightly better, due to its ability to overlap the startup cost of a send with the network latency of a previous message.

In Table 6–2, we present the execution times of the two implementations on a contention-free network.

We can see that the repeated-send implementation cannot take advantage of a network which can support multiple simultaneous messages, whereas the non-blocking version does so and scales much more effectively.

Nodes	PEXEC	
	Repeated send	*Non-blocking*
2	{(1101, 100%)}	{(1101, 100%)}
3	{(2202, 100%)}	{(2102, 100%)}
4	{(3303, 100%)}	{(3103, 100%)}
5	{(4404, 100%)}	{(4104, 100%)}
6	{(5505, 100%)}	{(5105, 100%)}
7	{(6606, 100%)}	-
8	{(7707, 100%)}	-

Table 6–1: Results of analysing scatter on a bus network

Nodes	PEXEC	
	Repeated send	*Non-blocking*
2	{(1101, 100%)}	{(1201, 100%)}
3	{(2202, 100%)}	{(1301, 100%)}
4	{(3303, 100%)}	{(1401, 100%)}
5	{(4404, 100%)}	{(1501, 100%)}
6	{(5505, 100%)}	{(1601, 100%)}
7	{(6606, 100%)}	{(1701, 100%)}
8	{(7707, 100%)}	{(1801, 100%)}

Table 6–2: Results of analysing scatter on a contention-free network

6.4 Modelling a Parallel Application

In this section we will model the performance of a scatter-gather application. First we will construct a model of the whole application and analyse its performance. We will then compare the accuracy and efficiency of an approach which models the scatter and gather separately and sums the analysed execution times to obtain a prediction of the whole application's performance.

6.4.1 Integrated Model

Our combined scatter-gather model defines processes as follows:

$$\text{Process}_i \stackrel{\text{def}}{=} \begin{cases} \overline{\text{send}_{10}}(d/p).\ldots.\overline{\text{send}_{p-1_0}}(d/p).\overline{\text{sent}_{10}}.\ldots.\overline{\text{sent}_{p-1_0}}. & (i=0) \\ \overline{\text{receive}_0}.\overline{\text{received}_0}.\ldots.\overline{\text{receive}_0}.\overline{\text{received}_0}.nil & \\ \\ \overline{\text{receive}_0}.\overline{\text{received}_0}.\overline{\text{send}_{00}}(d/p).\overline{\text{sent}_{00}}.nil & (i>0) \end{cases}$$

where d is the size of the dataset, and p is the number of processes. Table 6–3 presents the analysis of this system in the absence of contention, while Table 6–4 gives the results for the bus network. Note how the execution time falls in the contention-free case as message sizes decrease and can be overlapped, outweighing the cost of a larger number of message startups. On the bus network, overlapping cannot take place, so there is no reduction in execution time to offset the increased startup costs.

6.4.2 Micro-analysis Model

The micro-analysis model is composed of two operations: a scatter and a gather. The scatter is defined as follows:

Processes	PEXEC
2	$\{(1002002, 100\%)\}$
3	$\{(666668, 100\%)\}$
4	$\{(504002, 100\%)\}$

Table 6–3: Integrated scatter-gather model on contention-free system

Processes	PEXEC
2	$\{(1002002, 100\%)\}$
3	$\{ (1334336, 6.25\%),$
	$(1335335, 75\%),$
	$(1335336, 18.75\%)\}$
4	$\{ (1501006, 27.1\%),$
	$(1502004, 39.5\%),$
	$(1502005, 15.6\%),$
	$(1502006, 17.8\%)\}$

Table 6–4: Integrated scatter-gather model on bus network

Processes	PEXEC
2	$\{(501001, 100\%)\}$
3	$\{(335334, 100\%)\}$
4	$\{(253001, 100\%)\}$
5	$\{(204001, 100\%)\}$
6	$\{(171668, 100\%)\}$
7	$\{(148858, 100\%)\}$

Table 6–5: Scatter model on contention-free network

Processes	PEXEC
2	$\{(501001, 100\%)\}$
3	$\{(667668, 100\%)\}$
4	$\{(751003, 100\%)\}$
5	$\{(801004, 100\%)\}$

Table 6–6: Scatter model on bus network

$$\text{Process}_i \stackrel{\text{def}}{=} \begin{cases} \overline{\text{send}_{10}}(d/p). \ldots . \overline{\text{send}_{p-1_0}}(d/p). \overline{\text{sent}_{10}}. \ldots . \overline{\text{sent}_{p-1_0}}.nil & (i = 0) \\ \\ \overline{\text{receive}_0}. \overline{\text{received}_0}.nil & (i > 0) \end{cases}$$

Tables 6–5 and 6–6 presents the results of analysis in the absence and presence of contention respectively.

Our gather operation is defined by the following processes:

$$\text{Process}_i \stackrel{\text{def}}{=} \begin{cases} \overline{\text{receive}_0}. \overline{\text{received}_0}. \ldots . \overline{\text{receive}_0}. \overline{\text{received}_0}.nil & (i = 0) \\ \\ \overline{\text{send}_{00}}(d/p). \overline{\text{sent}_{00}}.nil & (i > 0) \end{cases}$$

The execution times obtained for gather on a contention-free network are given in Table 6–7. Table 6–8 presents execution times in the presence of contention.

Processes	PEXEC
2	{(501001, 100%)}
3	{(334334, 100%)}
4	{(251001, 100%)}

Table 6–7: Gather model on contention-free network

Processes	PEXEC
2	{(501001, 100%)}
3	{(667668, 100%)}
4	{(751003, 100%)}

Table 6–8: Gather model on bus network

By combining the scatter and gather results, we obtain an estimate of the whole application's performance, given in Tables 6–9 and 6–10.

6.4.3 Comparing Integrated and Micro-analysis Approaches

Comparing Tables 6–3 and 6–9 we see that the two approaches generate identical results for a contention-free environment.

A comparison of Tables 6–4 and 6–10 is more interesting. Contention between messages and acknowledgements leads to a distribution of execution times for the integrated model, while the micro-analysis approach gives a single execution time for each number of processes. The figure given by the micro-analysis method is, in fact, an upper bound on the execution times obtained from analysis of the integrated model. In general, sim-

Processes	*Total* PEXEC
2	{(1002002, 100%)}
3	{(669668, 100%)}
4	{(504002, 100%)}

Table 6–9: Micro-analysis total on contention-free network

Processes	*Total* PEXEC
2	$\{(1002002, 100\%)\}$
3	$\{(1335336, 100\%)\}$
4	$\{(1502006, 100\%)\}$

Table 6–10: Micro-analysis total on bus network

ilar results could be expected even in the absence of contention, depending upon the particular values of delays used. The micro-analysis method precludes any overlapping between the operations involved in consecutive phases of the application, and so gives an upper bound on execution time. The degree of inaccuracy introduced in this way will depend on the resources in use in consecutive phases. In the case where one phase made extensive use of a certain resource, say processors, and the next phase relied on a completely different resource, such as the network, an integrated model may well allow a significant amount of overlapping, while the micro-analysis approach would overestimate execution time. Where the same dominant resources are being contended for in consecutive phases, the micro-analysis approach may yield an acceptably tight upper bound.

Nodes	*Repeated send*			*Non-blocking*		
	States	*Time (s)*	*Memory (Mbyte)*	*States*	*Time (s)*	*Memory (Mbyte)*
2	15	0.6	-	15	0.6	-
3	34	1.2	-	35	1.3	-
4	62	3.2	-	64	4.6	1.3
5	110	8.4	1.8	113	14.0	2.0
6	202	23.2	3.5	206	43.0	3.0
7	390	67.3	7.5	395	106.7	8.1
8	786	197.7	12.3	792	283.4	13.0

Table 6–11: Scatter model resource requirements on contention-free system

Nodes	*Repeated send*			*Non-blocking*		
	States	*Time (s)*	*Memory (Mbyte)*	*States*	*Time (s)*	*Memory (Mbyte)*
2	15	0.6	-	15	0.6	-
3	34	1.2	-	64	4.0	-
4	62	3.2	-	270	21.6	3.3
5	110	8.4	1.8	1137	166.9	12.3
6	202	23.2	3.5	4649	1274.3	
7	390	67.3	7.5	-	-	-
8	786	197.7	12.3	-	-	-

Table 6–12: Scatter model resource requirements on bus network

6.5 Conclusions

The applicability of an analysis process is limited by its resource requirements. Tables 6–11 and 6–12 show the number of states visited, the analysis time, and the memory used in the analyses of scatter presented in Section 6.3.1. A SPARCstation 20 with 96 Mbyte of memory was used for these measurements.

Clearly, the number of states directly impacts the analysis time, but the memory re-

Processes	Integrated			Micro-analysis		
	States	*Time (s)*	*Memory (Mbyte)*	*States*	*Time (s)*	*Memory (Mbyte)*
2	56	1.6	-	72	2.1	-
3	156	7.0	1.5	222	13.0	1.6
4	322	32.8	4.1	1350	141.0	11.1

Table 6–13: Scatter-gather model resource requirements on contention-free system

quirements quickly grow to the point where larger systems cannot be analysed due to memory limitations.

The repeated send model is oblivious to any state in the network, so no increase in its resource requirements is observed when the possibility of contention is introduced into the network. The lack of state in the contention-free network means that the state space generated by the non-blocking model is only marginally larger than that of the repeated send implementation. However, a greater number of transitions generated by interleavings of action sequences still leads to longer analysis times.

Many modelling formalisms tackle burgeoning workloads associated with analysing complex models by decomposing the model into a number of smaller and more easily analysed sub-models, analysing these sub-models independently, and combining the results. In Section 6.4 we assessed the accuracy of a micro-analysis approach which can be thought of as a form of model decomposition. Tables 6–13 and 6–14 compare the resource requirements of the integrated model and the micro-analysis model. The figures quoted for the micro-analysis model are the sum of the number of states visited in each separate analysis, and the maximum amount of memory used by any single analysis. For the contention-free network, we see that the integrated model has markedly smaller resource requirements. This is primarily due to the very high number of possible interleavings which can arise at the start of a gather operation, which heavily skews the analysis requirements of the micro-analysis approach. In the integrated model, the times at which the various processes begin the gather operation are skewed by the preceding scatter, so the state space is not as large.

When we consider the same systems in the presence of contention, however, we see

Processes	*Integrated*			*Micro-analysis*		
	States	*Time (s)*	*Memory (Mbyte)*	*States*	*Time (s)*	*Memory (Mbyte)*
2	56	1.6	-	72	2.1	-
3	337	15.7	3.7	264	6.5	1.6
4	3365	327.9	22.6	1985	173.0	17.4

Table 6–14: Scatter-gather model resource requirements on bus network

that the micro-analysis approach has significantly lower resource requirements. Further investigation of decomposition techniques for model analysis is warranted.

The resource requirements of the analysis process preclude the study of large systems; the limitations of the approach make it better suited to being a validation tool which can be used to confirm intuition about models which are developed manually.

Chapter 7

Reasoning Formally About Parametric Models

We saw in Chapter 5 how we can derive a multi-set of execution times from an Eager Timed CCS agent which has concrete lower bounds on action times. In a parametric agent, concrete lower bounds are abstracted by time variables. A concrete agent can be thought of as an instance of a parametric agent at a particular point in its parameter space.

We can explore the parameter space of a parametric agent by selecting a number of points in parameter space then, for each point, instantiating the time variables in the parametric agent and analysing the execution times of the resultant concrete agent. This process of sampling the parameter space, shown in Figure 7–1, may involve substantial recalculation, and provides a purely numerical performance model which is not amenable to symbolic analysis. In this chapter we consider the alternative approach, also shown in Figure 7–1, of analysing the execution time of parametric agents directly and deriving a symbolic model of execution time. The symbolic execution time model can be manipulated directly, or instantiated with concrete values to provide concrete execution times.

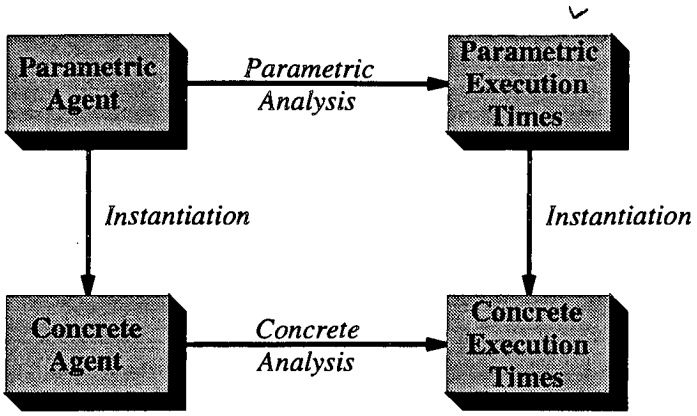


Figure 7–1: Reasoning about the execution time of concrete and parametric agents

7.1 Eager Timed CCS for Parametric Agents

In defining the labelled transition system $\stackrel{\alpha}{\Longrightarrow}_u$ for concrete agents, we relied on a number of conditions — comparisons between time expressions.

Definition 7.1.1 *The set $\mathcal{C}$ of* conditions *is defined as follows :*

$$\mathcal{C} ::= \top \mid \bot \mid e_1 \leq e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2$$

where $\top$ is always true, $\bot$ is always false, $e_1, e_2 \in \mathcal{E}_\mathcal{T}$ and $c_1, c_2 \in \mathcal{C}$. We write $e_1 < e_2$ to denote $(e_1 \leq e_2) \wedge (e_1 \neq e_2)$.

In the presence of variables, such comparisons, or conditions, cannot in general be evaluated, since they have different results in different regions of parameter space. We can think of a point in parameter space as a particular assignment of values to the time variables which describe the parameter space.

Definition 7.1.2 *A* valuation *$\mathcal{V} : V_\mathcal{T} \rightarrow \mathcal{T}$ is an assignment of values to time variables. We say that a valuation $\mathcal{V}$ satisfies a condition $c \in C$, written $\mathcal{V} \vdash c$, if the condition c evaluates to $\top$ for the assignment of values to variables defined by $\mathcal{V}$.*

A condition can be interpreted as a region of parameter space.

Definition 7.1.3 *The* region $\mathcal{R}(c)$ *of parameter space associated with a condition* $c \in \mathcal{C}$ *is defined as:*

$$\mathcal{R}(c) = \{\mathcal{V} | \mathcal{V} \vdash c\}$$

We will often be interested in whether there are *any* valuations which satisfy a condition.

Definition 7.1.4 *The* satisfiability predicate $\mathcal{S}(c)$ *is defined in terms of* $\mathcal{R}(c)$*:*

$$\mathcal{S}(c) \equiv \mathcal{R}(c) \neq \emptyset$$

We are interested in sets of objects, such as transitions or execution times, in which a condition is associated with each element of the set, determining the region of parameter space for which that element is a member of the set. We can think of this as a set of conditions labelled with objects.

Definition 7.1.5 *An* L-labelled condition set *is a set of* $L \times \mathcal{C}$ *pairs:*

$$\{(l_1, c_1), \ldots, (l_n, c_n)\}$$

We will sometimes require that no two conditions occurring in a labelled condition set can be satisfied simultaneously. A condition set with this property is said to be disjoint.

Definition 7.1.6 *In general, an L-labelled condition set* $\{(l_1, c_1), \ldots, (l_n, c_n)\}$ *is* disjoint *iff*

$$\forall i \in \{1, \ldots, n\}. \forall j \in \{1, \ldots, n\} - \{i\}. \mathcal{R}(c_i) \cap \mathcal{R}(c_j) = \emptyset$$

We also want condition sets to be minimal, or non-redundant, in the sense that there are no elements of the set whose conditions can never be satisfied.

Definition 7.1.7 *An L-labelled condition set* $\{(l_1, c_1), \ldots, (l_n, c_n)\}$ *is* non-redundant *if*

$$\forall i \in \{1, \ldots, n\}. \mathcal{R}(c_i) \neq \emptyset$$

An L-labelled condition set may be neither non-redundant nor disjoint. For example, the α-labelled condition set

$$A = \{(a, (x \leq 3) \wedge (4 \leq x)), (b, 0 \leq x), (c, 3 \leq x)\}$$

is not disjoint because

$$\mathcal{R}(0 \leq x) \cap \mathcal{R}(3 \leq x) \neq \emptyset$$

and it is not non-redundant because

$$\mathcal{R}((x \leq 3) \wedge (4 \leq x)) = \emptyset$$

Redundant elements can be removed straightforwardly by testing the satisfiability of their conditions. In general, however, a non-disjoint L-labelled condition set cannot be converted into an equivalent disjoint L-labelled condition set. Instead, we construct a $\mathbb{M}\ (L)$-labelled condition set which we call a partition. The key idea here is that producing a multiset of labels for each condition allows us to produce a disjoint set of of conditions while ensuring that each label of the original condition set is still associated with an equivalent set of conditions in the partition.

Definition 7.1.8 *A disjoint, non-redundant* $(\mathbb{M}\ (L))$*-labelled condition set* B *is a* partition *of an* L*-labelled condition set* A *iff*

1. *The region of parameter space associated with each label in the original set is exactly the same as the region of parameter space associated with that label in the partition:*

$$\forall (l, c) \in A. \bigcup \{\mathcal{R}(c_i) | (l, c_i) \in A\} = \bigcup \{\mathcal{R}(c_j) | (L_j, c_j) \in B \wedge l \in L_j\}$$

2. *The partition preserves the multiplicity of each label in the original condition set:*

$$\forall (L_i, c_i) \in B. \forall l \in L_i.$$
$$|\{x | (x \in L_i) \wedge (x = l)\}| = |\{c | ((l, c) \in A) \wedge (\mathcal{R}(c) \cap \mathcal{R}(c_i) = \mathcal{R}(c_i))\}|$$

Definition 7.1.9 presents a procedure for constructing a partition of an L-labelled condition set.

Definition 7.1.9 *The operator* Part : $\mathbb{F}\,(L \times \mathcal{C}) \rightarrow \mathbb{F}\,(\mathbb{M}\,(L) \times \mathcal{C})$ *is a partition. The operator is defined recursively:*

$$\begin{aligned} \text{Part}(\emptyset) &= \{(\emptyset, \top)\} \\ \text{Part}(\{x_1, \ldots, x_n\}) &= \text{Add}(x_n, \text{Part}(\{x_1, \ldots, x_n\}) \end{aligned}$$

Part *incrementally builds a partition by adding each element of the original labelled condition set in turn. Note that the initial partition* $\{(\emptyset, \top)\}$ *is both disjoint and non-redundant.*

The operator Add *defines the process of adding a new (value, condition) pair* (v, c) *to a partition*

$$P = \{(V_1, c_1), \ldots, (V_n, c_n)\}$$

We assume that $\mathcal{S}(c)$*, so* $\mathcal{R}(c) \neq \emptyset$*. If we consider adding* (v, c) *to each* (V_i, c_i) *in turn, we must address three cases:*

1. $\mathcal{R}(c) \cap \mathcal{R}(c_i) = \emptyset$

   In this case c *cannot be satisfied at the same time as* c_i*, so we need make no change to* (V_i, c_i)*. Since we have not changed the condition, we have preserved the disjoint and non-redundant properties required for a partition.*

2. $\mathcal{R}(c) \cap \mathcal{R}(c_i) = \mathcal{R}(c_i)$

   In this case c *can be satisfied everywhere that* c_i *holds, so we simply add* v *to* V_i*, replacing* (V_i, c_i) *with* $(V \cup \{\!|v|\!\}, c_i)$*. Again, since we have not changed the condition, we have preserved the disjoint and non-redundant properties required for a partition.*

3. $(\mathcal{R}(c) \cap \mathcal{R}(c_i) \neq \emptyset) \wedge (\mathcal{R}(c) \cap \mathcal{R}(c_i) \neq \mathcal{R}(c_i))$

In this case, c only holds in some parts of the region where c_i holds. We handle this by replacing (V_i, c_i) with (V_i, c_i') and (V_i, c_i'') where

$$\begin{aligned}\mathcal{R}(c_i') &= \mathcal{R}(c) \cap \mathcal{R}(c_i) \\ \mathcal{R}(c_i'') &= \mathcal{R}(c_i) - (\mathcal{R}(c) \cap \mathcal{R}(c_i))\end{aligned}$$

We then attempt to add (v, c) to (V_i, c_i') and (V_i, c_i'').

The operator Add *defines this process more succinctly:*

$$\begin{aligned}\mathrm{Add}((v, c_v), p) &= \{(V \cup \{\!|v|\!\}, c_V \wedge c_v) | (V, c_V) \in p \wedge \mathcal{S}(c_V \wedge c_v)\} \\ &\cup \{(V, c_V \wedge \mathrm{Neg}(c_v) | (V, c_V) \in p \wedge \mathcal{S}(c_V \wedge \mathrm{Neg}(c_v))\}\end{aligned}$$

Note that Add *relies on the operator* Neg *to determine c_i'' such that*

$$\mathcal{R}(c_i'') = \mathcal{R}(c_i) - (\mathcal{R}(c) \cap \mathcal{R}(c_i))$$

The operator $\mathrm{Neg} : \mathcal{C} \to \mathcal{C}$ *negates conditions:*

$$\begin{aligned}\mathrm{Neg}(\top) &= \bot && (1)\\ \mathrm{Neg}(\bot) &= \top && (2)\\ \mathrm{Neg}(e_1 < e_2) &= e_2 \leq e_1 && (3)\\ \mathrm{Neg}(e_1 = e_2) &= e_1 < e_2 \vee e_2 < e_1 && (4)\\ \mathrm{Neg}(c_1 \wedge c_2) &= \mathrm{Neg}(c_1) \vee \mathrm{Neg}(c_2) && (5)\\ \mathrm{Neg}(c_1 \vee c_2) &= \mathrm{Neg}(c_1) \wedge \mathrm{Neg}(c_2) && (6)\end{aligned}$$

We are now in a position to redefine $\mathcal{I}$, the set of intervals in which a parametric agent can perform a certain action as its next action. Definition 7.1.10 is an approximation which includes all legitimate intervals. However, it does not preclude actions which may be preempted, and it does not constrain the upper bounds of intervals due to preemption by other actions. We will see that this apparent relaxation is handled by eagerness in a later definition.

Definition 7.1.10 $\mathcal{I} : (\mathcal{P} \times \mathcal{C}) \times Act \to \mathbb{F}\,((\mathcal{E}_\mathcal{T} \times \mathcal{E}_\mathcal{T}) \times \mathcal{C})$ *gives the intervals in which a weakly guarded process can perform a certain action as its next action.*

$$
\begin{aligned}
\mathcal{I}((nil,c),\alpha) &= \emptyset && (1)\\
\mathcal{I}((a@t_v.P,c),a) &= \{((v,\infty),c)|\mathcal{S}(c)\} && (2)\\
\mathcal{I}((\tau@t_v.P,c),\tau) &= \{((v,v),c)|\mathcal{S}(c)\} && (3)\\
\mathcal{I}((\alpha@t_v.P,c),\beta) &= \emptyset \quad (\alpha \neq \beta) && (4)\\
\mathcal{I}((P+Q,c),\alpha) &= \mathcal{I}((P,c),\alpha) \cup \mathcal{I}((Q,c),\alpha) && (5)\\
\mathcal{I}((P|Q,c),a) &= \mathcal{I}((P,c),a) \cup \mathcal{I}((Q,c),a) && (6)\\
\mathcal{I}((P|Q,c),\tau) &= \mathcal{I}((P,c),\tau) && (7)\\
&\cup\ \mathcal{I}((Q,c),\tau)\\
&\cup\ \{\ ((u_1,u_1),c_P \wedge c_Q \wedge (u_3 < u_1) \wedge (u_1 \leq u_4))|\\
&\qquad \exists b \in \Lambda.\ (\exists((u_1,u_2),c_P) \in \mathcal{I}((P,c),b)).\\
&\qquad\qquad (\exists((u_3,u_4),c_Q) \in \mathcal{I}((Q,c),\overline{b})).\\
&\qquad\qquad \mathcal{S}((c_P \wedge c_Q \wedge (u_3 < u_1) \wedge (u_1 \leq u_4))\}\\
&\cup\ \{\ ((u_3,u_3),c_P \wedge c_Q \wedge (u_1 \leq u_3) \wedge (u_3 \leq u_2))|\\
&\qquad \exists b \in \Lambda.\ (\exists((u_1,u_2),c_P) \in \mathcal{I}((P,c),b)).\\
&\qquad\qquad (\exists((u_3,u_4),c_Q) \in \mathcal{I}((Q,c),\overline{b})).\\
&\qquad\qquad \mathcal{S}(c_P \wedge c_Q \wedge (u_1 \leq u_3) \wedge (u_3 \leq u_2))\}\\
\mathcal{I}((P\backslash L,c),a) &= \begin{cases} \emptyset & (a \in L \vee \overline{a} \in L)\\ \mathcal{I}((P,c),a) & (\text{otherwise}) \end{cases} && (8)\\
\mathcal{I}((P\backslash L,c),\tau) &= \mathcal{I}((P,c),\tau) && (9)\\
\mathcal{I}((P[f],c),a) &= \{((u_1,u_2),c')|\ \exists b \in \Lambda. && (10)\\
&\qquad\qquad f(b) = a\\
&\qquad\qquad ((u_1,u_2),c') \in \mathcal{I}((P,c),b)\}\\
\mathcal{I}((P[f],c),\tau) &= \mathcal{I}((P,c),\tau) && (11)\\
\mathcal{I}((\mu X.P,c),\alpha) &= \mathcal{I}((P\{\mu X.P/X\},c),\alpha) && (12)
\end{aligned}
$$

Using $\mathcal{I}$ we can obtain the conditional intervals in which an agent can perform a certain action as its next action. We are, of course, interested in all possible actions. In fact, we are interested in the set of possible actions for each region of parameter space.

Definition 7.1.11 *The function* $\delta : \mathcal{P} \times \mathcal{C} \rightarrow \mathbb{F}\ (\mathbb{F}\ (Act \times \mathcal{E}_\mathcal{T}) \times \mathcal{C})$ *gives a disjoint non-redundant* $(\mathbb{F}\ (Act \times \mathcal{E}_\mathcal{T}))$*-labelled condition set corresponding to the sets of possible (action, time expression) pairs in each region of parameter space.*

$$\delta(P, c_P) = \text{Part}(\bigcup_{\alpha \in Act}\{((\alpha, u_1), c_\alpha) | ((u_1, u_2), c_\alpha) \in \mathcal{I}((P, c_P), \alpha)\})$$

For each region of parameter space, $\delta(P, c_P)$ provides us with a set of possible (action, time expression) pairs.

Definition 7.1.12 *An (action, time expression) pair* (α, e_i) *with* $\alpha \in Act$ *and* $e \in \mathcal{E}_\mathcal{T}$ *is* dominated *in a set* $\{(\alpha_1, e_1), \ldots, (\alpha_n, e_n)\}$ *under condition* c *iff*

$$\forall \mathcal{V} \in \mathcal{R}(c). \exists j \in \{1, \ldots, n\}. \mathcal{V} \vdash e_j < e_i$$

We can remove dominated (action, time expression) pairs from the set of possible actions in a region, since eagerness guarantees that they cannot be performed as the next action.

Definition 7.1.13 *The function* $\Delta : \mathcal{P} \times \mathcal{C} \rightarrow \mathbb{F}\ ((Act \times \mathcal{E}_\mathcal{T}) \times \mathcal{C})$ *is the set of times and actions of the earliest possible actions which the given process can perform.*

$$\Delta(P, c_P) = \bigcup_{(E, c_E) \in \delta(P, c_P)} \Delta'(E, c_E)$$

where

$$\Delta'(E, c_E) = \bigcup_{(\alpha, u) \in E} \{((\alpha, u), c') |\ (c' = c_E \wedge (\bigwedge_{(\beta, v) \in E} u \leq v)) \wedge \mathcal{S}(c')\}$$

Now we want to allow the transition $P, c_P \stackrel{\alpha}{\Longrightarrow}_u X$ for some $X \in \mathcal{P} \times \mathcal{C}$ if $((\alpha, u,), c_\alpha) \in \Delta(P, c_P)$. It remains for us to define the derivative X. As we did for concrete agents, we will define two operators which return the derivatives of an agent reached by delaying and performing an action respectively.

Definition 7.1.14 Delay : $(\mathcal{P} \times \mathcal{C}) \times \mathcal{E}_{\mathcal{T}} \rightarrow \mathbb{F}\ (\mathcal{P} \times \mathcal{C})$ *gives the agent resulting from delaying the given weakly-guarded agent by the specified period of time.*

$$
\begin{array}{lcll}
\text{Delay}((nil, c), u) & = & \{(nil, c)\} & (1) \\
\text{Delay}((\alpha@t_v.P, c), u) & = & \{(\alpha@t_{v-u}.P\{u+t/t\}, c \wedge u < v) | \mathcal{S}(c \wedge u < v)\} & (2) \\
 & \cup & \{(\alpha@t_0.P\{u+t/t\}, c \wedge v \leq u) | \mathcal{S}(c \wedge v \leq u)\} & \\
\text{Delay}((P+Q, c), u) & = & \{(P'+Q', c_P \wedge c_Q) | \ (P', c_P) \in \text{Delay}((P, c), u) & (3) \\
 & & \qquad (Q', c_Q) \in \text{Delay}((Q, c), u) & \\
 & & \qquad \mathcal{S}(c_P \wedge c_Q)\} & \\
\text{Delay}((P|Q, c), u) & = & \{(P'|Q', c_P \wedge c_Q) | \ (P', c_P) \in \text{Delay}((P, c), u) & (4) \\
 & & \qquad (Q', c_Q) \in \text{Delay}((Q, c), u) & \\
 & & \qquad \mathcal{S}(c_P \wedge c_Q)\} & \\
\text{Delay}((P \backslash L, c), u) & = & \{(P' \backslash L, c_P) | (P', c_P) \in \text{Delay}((P, c), u) & (5) \\
\text{Delay}((P[f], c), u) & = & \{(P'[f], c_P) | (P', c_P) \in \text{Delay}((P, c), u) & (6) \\
\text{Delay}((\mu X.P, c), u) & = & \text{Delay}((P\{\mu X.P/X\}, c), u) & (7)
\end{array}
$$

Note that $\mu X.P$ must be weakly guarded, and that Delay$((P, c), t)$ *assumes that no action can take place before t has elapsed.*

Definition 7.1.15 Action : $(\mathcal{P} \times \mathcal{C}) \times Act \rightarrow \mathbb{M}\ (\mathcal{P} \times \mathcal{C})$ *gives the set of derivatives which can be reached from the given weakly-guarded agent by performing the specified action immediately.*

$$\text{Action}((nil, c), \alpha) = \{\!|\,|\!\} \tag{1}$$

$$\text{Action}((\alpha@t_v.P, c), \beta) = \begin{cases} \{\!|\ (P\{0/t\}, c \wedge v = 0)| & (\text{if } \alpha = \beta) \\ \quad \mathcal{S}(c \wedge v = 0)|\!\} & \\ \{\!|\,|\!\} & (\text{otherwise}) \end{cases} \tag{2}$$

$$\text{Action}((P + Q, c), \alpha) = \text{Action}((P, c), \alpha) \cup \text{Action}((Q, c), \alpha) \tag{3}$$

$$\text{Action}((P|Q, c), a) = \{\!|(P'|Q, c_P) \mid (P', c_P) \in \text{Action}((P, c), a)|\!\} \tag{4}$$
$$\cup\ \{\!|(P|Q', c_Q) \mid (Q', c_Q) \in \text{Action}((Q, c), a)|\!\}$$

$$\text{Action}((P|Q, c), \tau) = \{\!|(P'|Q, c_P) \mid (P', c_P) \in \text{Action}((P, c), \tau)|\!\} \tag{5}$$
$$\cup\ \{\!|(P|Q', c_Q) \mid (Q', c_Q) \in \text{Action}((Q, c), \tau)|\!\}$$
$$\cup\ \{\!|(P'|Q', c_P \wedge c_Q) \mid \exists b \in \Lambda.$$
$$(P', c_P) \in \text{Action}((P, c), b)$$
$$\wedge (Q', c_Q) \in \text{Action}((Q, c), \bar{b})$$
$$\wedge \mathcal{S}(c_P \wedge c_Q)|\!\}$$

$$\text{Action}((P\backslash L, c), a) = \begin{cases} \{\!|\,|\!\} & (\text{if } a \in L \vee \overline{a} \in L) \\ \{\!|\ (P'\backslash L, c_P)| & (\text{otherwise}) \\ \quad (P', c_P) \in \text{Action}((P, c), a)|\!\} & \end{cases} \tag{6}$$

$$\text{Action}((P\backslash L, c), \tau) = \{\!|(P'\backslash L, c_P)|(P', c_P) \in \text{Action}((P, c), \tau)|\!\} \tag{7}$$

$$\text{Action}((P[f], c), a) = \{\!|(P'[f], c_P)|\ \exists b \in \Lambda. \tag{8}$$
$$f(b) = a$$
$$(P', c_P) \in \text{Action}((P, c), b)|\!\}$$

$$\text{Action}((P[f], c), \tau) = \{\!|(P'[f], c_P)|(P', c_P) \in \text{Action}((P, c), \tau)|\!\} \tag{9}$$

$$\text{Action}((\mu X.P, c), \alpha) = \text{Action}((P\{\mu X.P/X\}, c), \alpha) \tag{10}$$

Now we are in a position to define the transition system of Eager Timed CCS for parametric agents.

Definition 7.1.16 *The labelled transition system of Eager Timed CCS for parametric agents is given by* $((\mathcal{P} \times \mathcal{C}), \{\stackrel{\alpha}{\Longrightarrow}_u | \alpha \in Act, u \in \mathcal{T}\})$ *where*

$$\stackrel{\alpha}{\Longrightarrow}_u = \{((P,c),(P',c'))|(\ (\alpha,u),c_\alpha) \in \Delta(P,c)$$
$$(P_D,c_D) \in \text{Delay}((P,c_\alpha),u)$$
$$(P',c') \in \text{Action}((P_D,c_D),\alpha)\}$$

The following examples illustrate the behaviour of parametric Eager Timed CCS agents.

Example 7.1.17 *The agent*

$$(a_t.nil, \top)$$

has a single transition

$$\stackrel{a}{\Longrightarrow}_t(nil, \top)$$

Example 7.1.18 *The agent*

$$(a_{t_1}.nil + \overline{a}_{t_2}.nil, \top)$$

has two possible derivatives

$$\stackrel{a}{\Longrightarrow}_{t_1}(nil, t_1 \leq t_2)$$

and

$$\stackrel{\overline{a}}{\Longrightarrow}_{t_2}(nil, t_2 \leq t_1)$$

Example 7.1.19 *The agent*

$$(a_{t_1}.nil|\overline{a}_{t_2}.nil, \top)$$

has three derivative sequences:

$$\stackrel{a}{\Longrightarrow}_{t_1}(nil|\overline{a}_{t_2-t_1}.nil, t_1 \leq t_2)\stackrel{\overline{a}}{\Longrightarrow}_{t_2-t_1}(nil|nil, t_1 \leq t_2)$$

$$\stackrel{\overline{a}}{\Longrightarrow}_{t_2}(a_{t_1-t_2}.nil|nil, t_2 \leq t_1)\stackrel{a}{\Longrightarrow}_{t_1-t_2}(nil|nil, t_2 \leq t_1)$$

and

$$\stackrel{\tau}{\Longrightarrow}_{t_1}(nil|nil, t_1 = t_2)$$

Example 7.1.20 *Adding a restriction to the previous example results in a different set of transitions. The agent:*

$$((a_{t_1}.nil|\overline{a}_{t_2}.nil)\backslash\{a\}, \top)$$

has two possible transitions:

$$\stackrel{\tau}{\Longrightarrow}_{t_2}((nil|nil)\backslash\{a\}, t_1 \leq t_2)$$

and

$$\stackrel{\tau}{\Longrightarrow}_{t_1}((nil|nil)\backslash\{a\}, t_2 < t_1)$$

Example 7.1.21 *The agent*

$$((a_{t_1}.nil|a_{t_1}.b_{t_2}.nil|\overline{a}_{t_3}.nil)\backslash\{a\}, \top)$$

has four derivative sequences:

$$\stackrel{\tau}{\Longrightarrow}_{t_3}((nil|a_0.b_{t_2}.nil|nil)\backslash\{a\}, t_1 \leq t_3)$$

$$\stackrel{\tau}{\Longrightarrow}_{t_1}((nil|a_0.b_{t_2}.nil|nil)\backslash\{a\}, t_3 < t_1)$$

$$\stackrel{\tau}{\Longrightarrow}_{t_3}((a_0.nil|b_{t_2}.nil|nil)\backslash\{a\}, t_1 \leq t_3)\stackrel{b}{\Longrightarrow}_{t_2}((a_0.nil|nil|nil)\backslash\{a\}, t_1 \leq t_3)$$

and

$$\stackrel{\tau}{\Longrightarrow}_{t_1}((a_0.nil|b_{t_2}.nil|nil)\backslash\{a\}, t_3 < t_1)\stackrel{b}{\Longrightarrow}_{t_2}((a_0.nil|nil|nil)\backslash\{a\}, t_3 < t_1)$$

7.2 Execution Time

We can redefine our notion of execution time for parametric models. Our rules are extended to take account of conditions:

$$\rightsquigarrow_1 \frac{}{(P,c) \overset{0}{\rightsquigarrow} (P,c)}(P \sim nil) \quad \rightsquigarrow_2 \frac{(P,c) \overset{\alpha}{\Longrightarrow}_{e_1} (Q,c') \;\; (Q,c') \overset{e_2}{\rightsquigarrow} (R,c'')}{(P,c) \overset{e_1+e_2}{\rightsquigarrow} (R,c'')}$$

and we redefine

$$\mathrm{EXEC}(P,c) = \{(e,c_Q) | (P,c) \overset{e}{\rightsquigarrow} (Q,c_Q)\}$$

Most commonly, we will be interested in $\mathrm{EXEC}(P,\top)$, but there may be circumstances in which we wish to make assumptions about the relationships between parameters, and consider only a subset of the parameter space using $\mathrm{EXEC}(P,c)$ for some $c \neq \top$.

Partitioning $\mathrm{EXEC}(P,c)$ gives us the set of execution time expressions corresponding to the possible executions in each region of parameter space. Given a set of expressions $\{e_1,\ldots,e_n\}$, we cannot in general determine the minimum or maximum expression if $n > 1$. We can, however, readily construct an expression for the average execution time:

$$\overline{e} = \frac{\sum_{i=1}^{n} e_i}{n}$$

There is one subtlety which arises in the presence of equality conditions. Consider the set $\{t_1, t_1, t_2 + t_3\}$, under condition $t_3 = t_1 - t_2$. A simple calculation of the average gives $(2t_1 + t_2 + t_3)/3$. Taking the equality constraint into account, however, we see that the average can be simplified to t_1. In general, dereferencing of time expressions with respect to equality conditions is a useful simplification technique.

We can define

$$\text{FAIREXEC}((P,c)) = \{\!|((u,p),c')|((P,c),1) \stackrel{u}{\rightsquigarrow} ((Q,c'),p)|\!\}$$

where $((P,c),p) \stackrel{t}{\rightsquigarrow} ((Q,c),p')$ is defined by the following rules:

$$\rightsquigarrow_1 \ \frac{}{((P,c),p) \stackrel{0}{\rightsquigarrow} ((P,c),p)}(P \sim nil)$$

$$\rightsquigarrow_2 \ \frac{(P,c) \stackrel{\alpha}{\Longrightarrow}_{u_1} (Q,c') \quad \left((Q,c'),\frac{p}{|l|}\right) \stackrel{u_2}{\rightsquigarrow} ((R,c''),p')}{((P,c),p) \stackrel{u_1+u_2}{\rightsquigarrow} ((R,c''),p')}$$

where $(l,c) \in \text{Part}(\Delta((P,c)))$ and $(\alpha,u_1) \in l$

We can easily partition the elements of $\text{FAIREXEC}((P,c))$ and combine the results to form a set rather than a multi-set:

$$\text{PEXEC}((P,c)) = \bigcup_{(l,c') \in \text{Part}(\text{FAIREXEC}((P,c)))} \{(\{(u, \sum_{(u,p)\in l} p) | (u,p) \in l\}, c')\}$$

The following examples show the execution times of Eager Timed CCS agents.

Example 7.2.1

$(a_t.nil, \top)$

$\stackrel{a}{\Longrightarrow}_t \ (nil, \top)$

$\text{EXEC}(a_t.nil, \top) = \{\!|(t, \top)|\!\}$

$\text{PEXEC}(a_t.nil, \top) = \{(\{(t, 100\%)\}, \top)\}$

Example 7.2.2

$(a_{t_1}.nil + \overline{a}_{t_2}.nil, \top)$

$\stackrel{a}{\Longrightarrow}_{t_1} \ (nil, t_1 \leq t_2)$

$\stackrel{\overline{a}}{\Longrightarrow}_{t_2} \ (nil, t_2 \leq t_1)$

$\text{EXEC}(a_{t_1}.nil + \overline{a}_{t_2}.nil, \top) = \{\!|(t_1, t_1 \leq t_2), (t_2, t_2 \leq t_1)|\!\}$

$$\mathrm{PEXEC}(a_{t_1}.nil + \overline{a}_{t_2}.nil, \top) = \{\ (\{(t_1, 100\%)\}, t_1 < t_2),\ (\{(t_1, 50\%), (t_2, 50\%)\}, t1 = t2),\ (\{(t_2, 100\%)\}, t_2 < t_1)\}$$

Example 7.2.3

$$(a_{t_1}.nil|\overline{a}_{t_2}.nil, \top)$$
$$\stackrel{a}{\Longrightarrow}_{t_1} \quad (nil|\overline{a}_{t_2-t_1}.nil, t_1 \leq t_2)$$
$$\stackrel{\overline{a}}{\Longrightarrow}_{t_2-t_1} \quad (nil|nil, t_1 \leq t_2)$$
$$\stackrel{\overline{a}}{\Longrightarrow}_{t_2} \quad (a_{t_1-t_2}.nil|nil, t_2 \leq t_1)$$
$$\stackrel{a}{\Longrightarrow}_{t_1-t_2} \quad (nil|nil, t_2 \leq t_1)$$
$$\stackrel{\tau}{\Longrightarrow}_{t_1} \quad (nil|nil, t_1 = t_2)$$

$$\mathrm{EXEC}(a_{t_1}.nil|\overline{a}_{t_2}.nil, \top) = \{|(t_2, t_1 \leq t_2), (t_1, t_2 \leq t_1), (t_1, t_1 = t_2)|\}$$

$$\mathrm{PEXEC}(a_{t_1}.nil|\overline{a}_{t_2}.nil, \top) = \{\ (\{(t_2, 100\%)\}, t_1 < t_2),\ (\{(t_1, 100\%)\}, t_2 < t_1),\ (\{(t_2, 100\%)\}, t_1 = t_2)\}$$

Example 7.2.4

$$((a_{t_1}.nil|\overline{a}_{t_2}.nil)\backslash\{a\}, \top)$$
$$\stackrel{\tau}{\Longrightarrow}_{t_2} \quad ((nil|nil)\backslash\{a\}, t_1 \leq t_2)$$
$$\stackrel{\tau}{\Longrightarrow}_{t_1} \quad ((nil|nil)\backslash\{a\}, t_2 < t_1)$$

$$\mathrm{EXEC}((a_{t_1}.nil|\overline{a}_{t_2}.nil)\backslash\{a\}, \top) = \{|(t_2, t_1 \leq t_2), (t_1, t_2 < t_1)|\}$$

$$\mathrm{PEXEC}((a_{t_1}.nil|\overline{a}_{t_2}.nil)\backslash\{a\}, \top) = \{\ (\{(t_2, 100\%)\}, t_1 < t_2),\ (\{(t_1, 100\%)\}, t_2 < t_1),\ (\{(t_1, 100\%)\}, t_1 = t_2)\}$$

Example 7.2.5

$$
\begin{aligned}
&((a_{t_1}.nil|a_{t_1}.b_{t_2}.nil|\overline{a}_{t_3}.nil)\backslash\{a\}, \top)\\
&\quad \stackrel{\tau}{\Longrightarrow}_{t_3} \quad ((nil|a_0.b_{t_2}.nil|nil)\backslash\{a\}, t_1 \leq t_3)\\
&\quad \stackrel{\tau}{\Longrightarrow}_{t_1} \quad ((nil|a_0.b_{t_2}.nil|nil)\backslash\{a\}, t_3 < t_1)\\
&\quad \stackrel{\tau}{\Longrightarrow}_{t_3} \quad ((a_0.nil|b_{t_2}.nil|nil)\backslash\{a\}, t_1 \leq t_3)\\
&\qquad\quad \stackrel{b}{\Longrightarrow}_{t_2} \quad ((a_0.nil|nil|nil)\backslash\{a\}, t_1 \leq t_3)\\
&\quad \stackrel{\tau}{\Longrightarrow}_{t_1} \quad ((a_0.nil|b_{t_2}.nil|nil)\backslash\{a\}, t_3 < t_1)\\
&\qquad\quad \stackrel{b}{\Longrightarrow}_{t_2} \quad ((a_0.nil|nil|nil)\backslash\{a\}, t_3 < t_1)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{EXEC}((a_{t_1}.nil|a_{t_1}.b_{t_2}.nil|\overline{a}_{t_3}.nil)\backslash\{a\}, \top) = \{\!| \; & (t_3, t_1 \leq t_3),\\
& (t_1, t_3 < t_1),\\
& (t_3 + t_2, t_1 \leq t_3)\\
& (t_1 + t_2, t_3 < t_1) |\!\}
\end{aligned}
$$

$$
\begin{aligned}
\{ \; & \mathrm{PEXEC}((a_{t_1}.nil|a_{t_1}.b_{t_2}.nil|\overline{a}_{t_3}.nil)\backslash\{a\}, \top) =\\
& (\{(t_1, 50\%), (t_1 + t_2, 50\%)\}, t_3 < t_1),\\
& (\{(t_3, 50\%), (t_3 + t_2, 50\%)\}, t_1 \leq t_3)\}
\end{aligned}
$$

7.3 Automated Analysis of Execution Time

The major additional issue introduced to automated analysis by parametric agents is the need to test for the satisfiability of conditions. For example, we would not want the agent

$$a_{2x}.(c_x.nil + d_3.nil) + b_3.nil$$

to be capable of the execution sequence $\stackrel{a}{\Longrightarrow}_{2x}\stackrel{d}{\Longrightarrow}_3 nil$ since we require $2x \leq 3$ for $\stackrel{a}{\Longrightarrow}_{2x}$ to be possible, while $\stackrel{d}{\Longrightarrow}_3$ requires $3 \leq x$, and both cannot hold for any x. The predicate $\mathcal{S}$ would preclude this execution, allowing only $\stackrel{c}{\Longrightarrow}_x$ after $\stackrel{a}{\Longrightarrow}_{2x}$.

In general our conditions are arbitrary logical expressions involving disjunctions and conjunctions of linear constraints. Our first step in testing the satisfiability of a condition is to rearrange it into an equivalent condition in disjunctive normal form.

Definition 7.3.1 *A condition c is in* disjunctive normal form *(DNF) if $c \equiv c_1 \vee \ldots \vee c_n$ where $c_1, \ldots, c_n$ are conjunctions of literals.*

We can test the satisfiability of a condition $c \equiv c_1 \vee \ldots \vee c_n$ in DNF by testing the satisfiability of its clauses which, by definition, are each conjunctions of linear constraints. For c to be unsatisfiable, all c_i $(1 \leq i \leq n)$ must be unsatisfiable. To show that c is satisfiable, we need only show that one conjunction c_i $(1 \leq i \leq n)$ is satisfiable. In general, converting constraints into DNF will provide us with a number of alternative conjunctions of linear constraints to test for satisfiability. We can either test all of them, and record a list of solved forms for each state, or record the first successfully tested one, and backtrack should unsatisfiability be detected at a derivative state.

7.3.1 Constraint Satisfiability

The constraint logic programming has developed efficient techniques for determining the satisfiability of conjunctions of constraints. In the absence of disequations (and therefore strict inequalities), the classical technique for testing the satisfiability of a set of constraints over non-negative rational or real variables is the simplex method [Dantzig *et al.*, 1954]. The worst-case computational complexity of the simplex method is exponential but the average case performance has been found to be acceptable for many applications.

Given a set of linear equations defining a simplex in parameter space, and an objective function, the simplex method seeks to determine the set of assignments to parameters which minimises the objective function within the simplex. In fact, the simplex method needs an initial point, or basis, which satisfies the equations without necessarily minimising the objective function. However, by recasting the problem, the simplex algorithm can be used to determine its own starting point. This is done by adding a unique arti-

ficial variable z_i to each equation, and then using the simplex method to minimise the objective function $\sum_i z_i$. If the minimum of this function is greater than zero, then the original set of equations is not satisfiable. If, on the other hand, the objective function has a minimum at zero, then we know that the original set of equations is satisfiable, and furthermore, we can extract a set of parameters which satisfy the original equations from the results of the simplex algorithm, which gives us the input we needed for the original problem. We can now use the simplex algorithm a second time, this time with the original objective function, to obtain the optimum parameter values.

If we express our constraint set as a set of linear equations, we can use the first phase of the simplex method to determine the satisfiability of the constraints. We are not interested in the optimum parameter values, indeed we do not have an objective function in mind, but rather we want to know whether any set of parameters exists which satisfies the constraints. Each equality constraint is already an equation and needs no manipulation. We replace each non-strict inequality $c \equiv e_1 \leq e_2$ by an equation $e_1 + s = e_2$ in which a slack variable s_i has been introduced. Having produced our set of equations, we can use the first phase of the simplex method to determine the satisfiability of the constraint set.

The simplex method actually maps any satisfiable set of linear equations into a solved form, which we refer to as SF1.

Definition 7.3.2 *A linear equation*

$$y = a_0 + \sum_{i=1}^{n} a_i x_i$$

where $\forall i \in \{1, \ldots, n-1\}.x_i \gg x_{i+1}$ *(given some ordering on variables) is defined to be in* SF1 *iff*

1. $\forall i \in \{1, \ldots, n | a_i \neq 0\}.x_i \neq y$;

2. $a_0 \geq 0$

y is said to be the basic variable of the equation while the variables

$$\bigcup_{i \in \{1,\ldots,n\}} \{x_i | a_i \neq 0\}$$

are said to be the non-basic variables. If there are no non-basic variables in a linear equation in SF1, it is called an explicit assignment. A set of linear equations is in SF1 if each of the equations is in SF1, and each basic variable appears only once.

7.3.2 The problem with disequations

To reason about parametric agents in Eager Timed CCS, we require a method which copes with strict inequalities, and therefore disequations. Van Hentenryck and Graf [Van Hentenryck and Graf, 1992] define a new solved form SF2 by generalising SF1 to handle disequations, and therefore strict inequalities.

Definition 7.3.3 *A disequation*

$$0 \neq a_0 + \sum_{i=1}^{n} a_i x_i$$

where $\exists i \in \{0, \ldots, n\}.a_i \neq 0$.

is defined to be in SF2.

A set $< \mathcal{E}, \mathcal{D} >$ *of linear equations* $\mathcal{E}$ *and disequations* $\mathcal{D}$ *is defined to be defined to be in SF2 if*

1. $\mathcal{E}$ *is in SF1;*

2. *every disequation in* $\mathcal{D}$ *is in SF2;*

3. *and the disequations in* $\mathcal{D}$ *do not contain any basic variables of* $\mathcal{E}$.

Van Hentenryck and Graf observe that disequations are passive, in the sense that they cannot be used to determine the value of a variable, only to exclude certain parts of the

solution space. Using a geometric interpretation, the forbidden space associated with a disequation d is a hyperplane $\mathcal{H}$, and given the polyhedral set $\mathcal{P}$ defined by a set of equations $\mathcal{E}$, three cases are possible:

1. $\mathcal{P} \subseteq \mathcal{H}$ in which case there is no solution

2. $\mathcal{P} \cap \mathcal{H} = \emptyset$ in which case the disequation does not preclude any of the solution space

3. $\mathcal{P} \cap \mathcal{H} \neq \emptyset$ and $\mathcal{P} \subseteq \mathcal{H}$ in which case there is a solution, but the disequation excludes part of the solution space.

SF2 is intended to avoid case 1 arising, and assuming that all variables can be assigned at least two values, this holds. Van Hentenryck and Graf point out that, unfortunately, SF2 can have implicit constants, which do not appear in an explicit assignment. Consider, for example, the set of constraints:

$$y_1 = 0 + x_1 - x_2$$
$$y_2 = 0 - x_1 + x_2$$
$$0 \neq x_1 - x_2$$

While this set of constraints is in SF2, y_1 and y_2 can only be assigned the value 0 (recall that both are constrained to be non-negative). However, this assignment is prevented by the disequation. Hence a system of linear equations and disequations which is not satisfiable can be expressed in SF2, so SF2 cannot be regarded as a solved form.

7.3.3 Handling disequations

Lassez and McAloon [Lassez and McAloon, 1988] propose a canonical form for linear constraints which consists of three sets of constraints and propose algorithms, based on optimisation procedures, to remove implicit equalities (and hence hidden constants) and

to eliminate redundant constraints from sets of constraints. This has broad scope, tackling the standardisation of output from constraint solvers, and the equivalence of systems of constraints. However, the potentially very expensive optimisation approaches may outweigh the benefits of the canonical form.

Stuckey [Stuckey, 1990] proposes a perturbation technique, which involves modifying the constraints slightly to determine whether the unsatisfiability is "real" or due to implicit inequalities. If implicit inequalities are held to be responsible, a special procedure is used to identify them.

A more natural approach [Van Hentenryck and Graf, 1992] proposes a new standard form, SF3, which precludes hidden constants. SF3 is based on the notion of lexicographically positive vectors.

Definition 7.3.4 *A vector $\overline{v}$ is lexicographically positive (respectively negative), denoted $\overline{v} \overset{L}{>} 0$ (respectively $\overline{v} \overset{L}{<} 0$), if its first non-zero component is positive (respectively negative). We will write $\overline{v} \overset{L}{\geq} 0$ in place of the disjunction $\overline{v} = 0 \vee \overline{v} \overset{L}{>} 0$. A vector $\overline{v}_1$ is said to be lexicographically greater than a vector $\overline{v}_2$ if $\overline{v}_1 - \overline{v}_2 \overset{L}{>} 0$. A negative unit vector is a vector $(0, \ldots, 0, -1, 0, \ldots, 0)$.*

Van Hentenryck and Graf associate a vector $(a_0, \ldots, a_{k-1}, -1, a_k, a_n)$ with each equation:

$$y = a_0 + \sum_{i=1}^{n} a_i x_i$$

where $x_{k-1} \gg y \gg x_k$.

Definition 7.3.5 *A linear equation is considered to be in SF3, if it is in SF1, and the associated vector is either a negative unit vector (to handle $y = 0$) or lexicographically positive. A set of equations is considered to be in SF3 if each equation is in SF3 and each basic variable appears only once.*

A linear disequation $0 \neq t$ is in SF3 iff it is in SF2 and the vector associated with t is lexicographically positive.

A set $< \mathcal{E}, \mathcal{D} >$ of linear equations and disequations is in SF3 if:

1. *$\mathcal{E}$ is in SF3;*

2. *all disequations in $\mathcal{D}$ are in SF3;*

3. *every disequation in $\mathcal{D}$ does not contain basic variables of $\mathcal{E}$.*

Van Hentenryck and Graf show that a system of equations and disequations in SF3 cannot include hidden constants, and that it is satisfiable if and only if it can be mapped into SF3. They go on to modify the simplex algorithm to preserve SF3 through pivoting. Failure to preserve SF3 would require an explicit search for hidden constants each time a new constraint is introduced, which would be inefficient.

Pivoting a set of equations $\mathcal{E}$ in standard form amounts to transforming $\mathcal{E}$ into an equivalent $\mathcal{E}'$ where exactly one non-basic variable has been turned into a basic variable and vice versa. An iteration of the simplex algorithm consists of three steps:

1. choose a non-basic variable to enter the basis ("the entering variable");

2. choose a basic variable to leave the basis ("the leaving variable")

3. Pivot to remove the leaving variable from and introduce the entering variable to the basis.

Van Hentenryck and Graf modify the second step. Given

$$\mathcal{E} \equiv \left\{ y_i = a_{i0} + \sum_{j=1}^{n} a_{ij} x_j \,\middle|\, (1 \leq i \leq n) \right\}$$

and an entering variable x_k, the simplex algorithm chooses the leaving variable y_l such that

$$\frac{a_{i0}}{alk} = \max_{i \in S_k} \frac{a_{i0}}{a_{ik}}$$

where

$$S_k = \{i | (1 \leq i \leq m) \wedge (a_{ik} < 0)\}$$

The simplex algorithm does not prescribe a specific rule to break ties. Van Hentenryck and Graf use the lexicographic rule to do so. If v_i is the vector associated with constraint i, then we define u_i to be v_i/a_{ik}. The lexicographic rule amounts to choosing y_l in such a way that u_l is the lexicographically greatest of all vectors, i.e.

$$\neg\exists u \in \bigcup_{i \in \{1,\ldots,m\}} \{v_i/a_{ik}\}.u \gg u_l$$

Van Hentenryck and Graf show that the lexicographic rule preserves SF3 through pivoting, and argue that it is of great practical interest since it introduces only a slight overhead over the simplex algorithm. Moreover, the lexicographic pivoting rule was originally proposed to prevent cycling, so it also ensures that the simplex algorithm terminates.

7.3.4 An incremental algorithm

Finally, Van Hentenryck and Graf propose an algorithm for incremental problems. This is of interest in reasoning about parametric Eager Timed CCS agents, since we will accumulate constraints as we traverse state space, and a method which exploits the solved form of constraints in the previous state will avoid unnecessary repeated work. Given a system $\mathcal{E}$ in SF3 to which a new constraint C is to be added the steps of the algorithm are:

1. Remove the basic variables of $\mathcal{E}$ from C through dereferencing to obtain C';

2. Rewrite C' into the form

$$0 = a_0 + \sum_{i=1}^{n} a_i x_i$$

which is lexicographically positive, possibly by multiplying by -1. We ignore the trivial case $0 = 0$. There are now two cases to consider:

(a) If $a_0 = 0$ then all variables x_i with $a_i > 0$ are equal to 0. This should be propagated into $\mathcal{E}$ giving $\mathcal{E}'$, which must be checked to see if it is still in SF3. This is done by removing any constraint which is not in SF3 from $\mathcal{E}'$ and re-introducing them using this algorithm. This process is guaranteed to terminate because removing an implicit equality decreases the number of variables.

(b) Otherwise, we introduce an artificial variable z:

$$z = a_0 + \textstyle\sum_{i=1}^{n} a_i x_i$$

and minimize z, as is usual in the first stage of the simplex algorithm. If $z = 0$, we can remove z from the system, which might involve removing it from the basis. There is no need to check for implicit equalities since SF3 is preserved through pivoting.

3. When we add an equation, we have to reconsider the disequations. For each disequation in turn, we dereference the disequation with respect to the basic variables of the solved form, giving a new disequation of the form $0 \neq t$. If t is 0 then the new set of constraints is not satisfiable, otherwise the disequation can be mapped easily into SF3

Adding a disequation is straightforward. We dereference it with respect to the basic variables of the solved form, obtaining $0 \neq t$. Again, if t is 0 the constraint set is not satisfiable, otherwise, it is easily mapped into SF3.

7.3.5 A better incremental algorithm

Van Hentenryck and Imbert [Van Hentenryck and Imbert, 1993] show that disequations can be processed more efficiently by isolating subclasses of equations which, when added, do not require the disequations to be checked, and/or by specialising SF3 to isolate various subclasses of equations and disequations. They begin by differentiating between arbitrary and slack variables. They split $\mathcal{E}$ into $\mathcal{E}_s \cup \mathcal{E}_a$ where $\mathcal{E}_s$ is the set of equations over only slack variables, and $\mathcal{E}_a$ is the set of equations which include at least one arbitrary variable. $\mathcal{D}$ is similarly split into $\mathcal{D}_s$ and $\mathcal{D}_a$. SF3 is then redefined in terms of the four sets $\mathcal{E}_a$, $\mathcal{E}_s$, $\mathcal{D}_a$ and $\mathcal{D}_s$.

Van Hentenryck and Imbert go on to show how the solved form can be specialised further, permitting more efficient algorithms for adding constraints. The basic idea is to divide the sets $\mathcal{E}_a$ and $\mathcal{D}_a \cup \mathcal{D}_s$ depending on the number of arbitrary and/or slack variables occurring in the constraint. Three subclasses are identified:

1. The first subclass of equations worth exploiting is where the equation in solved form includes at least one arbitrary variable in its right hand side. Since we are considering equations in $\mathcal{E}_a$, each equation has an arbitrary variable as its basic variable. Recall that one of the major purposes of the solved form is to recognize implicit constants. It is guaranteed that the basic variable is not fixed, since an arbitrary variable occurs on the right hand side. Hence any slack variables occurring in the right hand side of the equation do not need to be dereferenced with respect to $\mathcal{E}_s$, sparing computation. This subclass of equations is important since slack variables are always introduced by inequalities, producing a constraint of this type.

2. A second subclass of equations which can be exploited has precisely one slack variable and no arbitrary variables on the right hand side. By definition of the solved form, this slack variable is not fixed (implying that the arbitrary basic variable is not fixed), and hence the slack variable does not need to be dereferenced. It is only necessary to reconsider this constraint when the slack variable becomes

fixed. Again, this subclass is a frequent case, generated by common constraints of the form $x \geq a$ and $x \leq s$.

3. A third subclass is the set of equations which have exactly one arbitrary variable and no slack variables on the right hand side. Provided that the arbitrary variable is not fixed, and does not introduce a dereferencing loop in this subclass, there is no need to dereference it.

These three criteria lead to $\mathcal{E}_a$ being divided into five subclasses $\mathcal{E}_F$, $\mathcal{E}_{A1}$, $\mathcal{E}_{A2}$, $\mathcal{E}_{C1}$ and $\mathcal{E}_{C2}$, each with an invariant which must be satisfied by all the members of that class.

Class	*RHS Variables*		*Invariant*
	Arbitrary	*Slack*	
$\mathcal{E}_F$	0	0	-
$\mathcal{E}_{A1}$	1	0	dereferenced w.r.t. $\mathcal{E}_F \cup \mathcal{E}_{A1}$
$\mathcal{E}_{A2}$	$i \geq 1$	$j \geq \max(0, 2-i)$	dereferenced w.r.t. $\mathcal{E}_a$
$\mathcal{E}_{C1}$	0	1	No fixed slack variable w.r.t. $\mathcal{E}_s$
$\mathcal{E}_{C2}$	0	≥ 2	dereferenced w.r.t. $\mathcal{E}_s$

There are three exactly symmetrical cases for disequations.

1. In the first subclass, we consider disequations which include at least one arbitrary variable. Such disequations are trivially satisfiable since an arbitrary variable can take on any value whatever the values assigned to other variables in the disequation. Hence any slack variables in the disequation need not be dereferenced.

2. In the second subclass, the disequation contains precisely one slack variable and no arbitrary variables. The slack variable does not need to be dereferenced with respect to $\mathcal{E}_s$ and this disequation only needs to be checked when the slack variable becomes fixed.

3. The third subclass contains disequations in which exactly one arbitrary variable is present. We need reconsider such a disequation only when its arbitrary variable becomes fixed.

These lead to four subclasses of $\mathcal{D}_a \cup \mathcal{D}_s$:

Class	*Variables*		*Invariant*
	Arbitrary	*Slack*	
$\mathcal{D}_{A1}$	1	0	No fixed arbitrary variable w.r.t. $\mathcal{E}_a$
$\mathcal{D}_{A2}$	$i \geq 1$	$j \geq \max(0, 2-i)$	dereferenced w.r.t. $\mathcal{E}_a$
$\mathcal{D}_{C1}$	0	1	No fixed slack variable w.r.t. $\mathcal{E}_s$
$\mathcal{D}_{C2}$	0	≥ 2	dereferenced w.r.t. $\mathcal{E}_s$

Van Hentenryck and Imbert go on to present complex and subtle algorithms for adding equations and disequations which preserve the solved form, while avoiding unnecessary work.

Van Hentenryck and Imbert's algorithms have been implemented as part of an analysis tool for parametric agents. This tool is loosely based on the analysis tool for concrete agents presented in Chapter 5, but the need to take account of constraints at the lowest levels required extensive re-implementation of the original version, as well as the addition of substantial modules for constraint satisfaction.

The following examples show the tool in use.

Example 7.3.6 *The results of analysing the file:*

```
define P1 = (a,t).nil
analyse P1
```

are:

```
2 states
Summary:
  t      (100.0%)  (((<T>)))
```

Example 7.3.7 *The results of analysing the file:*

```
define P2 = (a,t1).nil + (a',t2).nil
analyse P2
```

are:

```
3 states
Summary:
  t2     (50.0%)  (((t1-t2>=0) & (-t1+t2>=0)))
  t1     (50.0%)  (((t1-t2>=0) & (-t1+t2>=0)))

  t1     (100.0%)  (((-t1+t2>0)))

  t2     (100.0%)  (((t1-t2>0)))
```

Example 7.3.8 *Analysing the file:*

```
define P3 = (a,t).nil + (a,t).nil
analyse P3
```

yields:

```
2 states
Summary:

  t      (100.0%)  (((<T>)))
```

Example 7.3.9 *The file:*

```
define P4 = (a,t1).nil|(a',t2).nil
analyse P4
```

produces the following analysis:

```
8 states
Summary:
  t1     (100.0%)  (((t1-t2>0)))
```

```
t2     (100.0%)  (((-t1+t2>0)))

t2     (100.0%)  (((-t1+t2=0)))
```

Example 7.3.10 *Analysing the file:*

```
define P5 = (a,t).nil|(a',t).nil
analyse P5
```

yields:

```
4 states
Summary:

  t     (100.0%)  (((<T>)))
```

Example 7.3.11 *The file:*

```
define P6 = ((a,t1).nil|(a',t2).nil)\{a}
analyse P6
```

produces:

```
4 states
Summary:

  t1     (100.0%)  (((t1-t2>0)))

  t2     (100.0%)  (((-t1+t2>0)))

  t1     (100.0%)  (((-t1+t2=0)))
```

Example 7.3.12 *Analysing:*

```
define P7 = ((a,t1).nil|(a,t1).(b,t2).nil|
            (a',t2).nil)\{a}
analyse P7
```

results in:

```
10 states
Summary:
  t1     (50.0%)  (((t1-t2>0)))
  t1+t2  (50.0%)  (((t1-t2>0)))

  t2     (50.0%)  (((-t1+t2>0)))
  +2t2   (50.0%)  (((-t1+t2>0)))

  t1     (50.0%)  (((-t1+t2=0)))
  t1+t2  (50.0%)  (((-t1+t2=0)))
```

Example 7.3.13 *The agent described by the file:*

```
define P8 = ((a,t).nil|(b,t).nil|(c,t).nil|(d,t).nil)
analyse P8
```

produces the analysis output:

```
16 states
Summary:

  t      (100.0%)  (((<T>)))
```

Example 7.3.14 *Analysing the agent described by file:*

```
define P9 = ((a,t).nil)[b/a]
analyse P9
```

produces:

```
2 states
Summary:

  t      (100.0%)  (((<T>)))
```

Example 7.3.15 *Analysing the agent:*

```
define P10 = (((a,t1).nil)[b/a]|(b',t2).nil)\{b}
analyse P10
```

yields:

```
4 states
Summary:

  t1     (100.0%)  (((t1-t2>0)))

  t2     (100.0%)  (((-t1+t2>0)))

  t1     (100.0%)  (((-t1+t2=0)))
```

Example 7.3.16 *The restricted parallel composition:*

```
define P11 = ((b',t1).nil|(a,t2).nil)\{b}
analyse P11
```

has the following analysis results:

```
2 states

Summary:

  t2      (100.0%)  (((<T>)))
```

The time taken to analyse the above examples on a SPARCstation 20 with 96 Mbyte of RAM, along with details of the number of variables and states involved in each case, is shown in Table 7–1.

Example	*States*	*Variables*	*Time (s)*
7.3.6	2	1	0.0
7.3.7	3	2	0.1
7.3.8	2	1	0.0
7.3.9	8	2	0.7
7.3.10	4	1	0.0
7.3.11	4	2	0.3
7.3.12	10	2	3.3
7.3.13	16	1	0.1
7.3.14	2	1	0.0
7.3.15	4	2	0.3
7.3.16	2	2	0.0

Table 7–1: Analysis time and parameters for parametric agents

The analysis time does not depend simply on the number of states. Indeed, comparing Examples 7.3.12 and 7.3.13 we see a reduction in execution time, despite a marked increase in the number of states. This occurs because analysis time is dominated by constraint satisfaction, which is heavily influenced by the number of constraints to be tested and their complexity. This is driven by the number of variables and the structure of the agent being analysed. As the number of variables and the structural complexity of

the agent is increased, the memory required for analyses grows very rapidly and quickly exceeds the memory available on the platform used, even for very small examples.

7.4 Further Work

The analysis tool can derive symbolic expressions for the execution time of a concurrent system in different regions of its parameter space. However, there is significant scope for improving the efficiency of the current tool, thereby enabling the analysis of more realistic systems.

The existing constraint solver could be replaced with a more fully incremental implementation of the algorithms presented in [Van Hentenryck and Imbert, 1993]. This would avoid considerable repeated computation which the current implementation performs, potentially yielding substantial performance improvements. More importantly, a more compact representation of constraints, based on the solved form, which could be shared by the transition rules, the constraint satisfaction engine and the output module would reduce memory requirements, which are currently the limiting factor for the analysis of larger and more realistic systems.

The methods used to identify redundancy and simplify constraint sets for output could also be improved. The current implementation does not maintain constraint sets in a minimal form, so satisfiability tests are used to simplify a constraint set before output. These tests are expensive, and savings could be made by replacing this ad hoc approach with methods from the constraint logic programming community.

A related but more demanding issue is that of combining partitions of parameter space. The conditional execution times

$$(t_1, t_1 \geq t_2)$$

and

$$(t_2, t_2 > t_1)$$

can clearly be combined into

$$(\max\{t_1, t_2\}, \top)$$

General techniques for this form of expression simplification seem essential prerequisites for the effective analysis of large systems.

Chapter 8

Conclusions and Further Work

Symbolic models of execution time can be used to integrate performance considerations into the design process. In this thesis, we have explored methods for developing symbolic models of parallel applications' execution times. We have assessed the accuracy with which micro-analysis models can predict the execution time of sequential code fragments, collective communication operations and parallel application structures. We have demonstrated a number of uses of symbolic performance models in the design process, and identified difficulties in developing such models manually. We have developed techniques for formally deriving symbolic expressions of execution time from models of concurrent systems.

In doing so, we have extended previous work in this area in four main ways. Firstly, we have assessed micro-analysis techniques for sequential performance prediction in a more challenging environment than previous studies, using sophisticated microprocessors and aggressive optimising compilers. Secondly, we have used collective communication operations, rather than pair-wise message-passing, as primitives in our micro-analysis of parallel applications and we have characterised the performance of these operations both symbolically and numerically for a range of platforms. Thirdly, we have proposed a novel formalism, based on a timed process calculus, for analysing the performance of parallel applications, and implemented an automatic analysis tool. Finally, we have extended this formalism to support abstract, parametric models so that we can automatically derive symbolic, rather than numerical, models of performance.

In Chapter 2 we used micro-analysis techniques to predict the execution time of sequential code fragments. Our experiments were conducted in a more challenging environment than many previous studies on micro-analysis. Performing the experiments on super-scalar microprocessors introduced greater potential for concurrent and overlapped execution than on simpler processor architectures. Nevertheless, we predicted the execution time of 6 code fragments on 3 platforms to within a factor of 1.8. This is an improvement over simple Mflop/s-based calculations, which predicted the code fragments' execution times to within a factor of 2.5. The use of Fortran introduced more sophisticated compiler optimisations than are found in compilers for many other languages. Enabling these optimisations increased the differences between predicted and measured performance significantly. For all platforms, the micro-analysis predictions were within a factor of 3.6 of measured performance, which is more accurate than Mflop/s calculations which result in predictions within a factor of 5.7. For two of the three platforms, micro-analysis predictions lay within a factor of at most 1.7 from the measured performance. In the case of the third platform, very much larger differences were observed. Improving the accuracy of our predictions requires taking greater account of the memory hierarchy and the sequence in which operations are performed. It would also be interesting to explore the extent to which a larger number of basic operations could be used to characterise more accurately the differences between programs, and basic operation times could be more accurately derived from a more representative range of code fragments.

In Chapter 3 we modelled the performance of a number of collective communication operations on a range of parallel computing platforms. Previous work on characterising the performance of interprocess communication has concentrated on primitive pair-wise message-passing operations, in the absence of contention. In working with collective communication operations, this research is closer to current practice in parallel software development. It also represents a more challenging environment for performance prediction, since there is considerable potential for interaction and contention between the pair-wise operations which implement each collective communication operation. We measured the performance of a range of collective communication operations on a num-

ber of configurations of several representative parallel computing platforms. Simple symbolic models were fitted to the data using multi-linear regression techniques. These models very accurately characterised the execution time of collective operations on the Cray T3D, with average prediction differences for particular collective operations ranging from 2% to 21%. The predictions for the other platforms considered were less accurate, ranging from 4% to 60%. This reduction in accuracy can be attributed to architectural features which led to greater potential for contention in the platforms in question, and to wide variation in measurements. The use of a dedicated cluster of workstations could help determine whether the large variation in the measured execution time of collective operations is intrinsic in the protocol stacks being used, or a function of interference with other traffic on the network. Despite the poor quality of some of the experimental data, we predicted the execution time of communication structures at least as accurately as we predicted the execution time of code fragments in Chapter 2.

Using techniques from Chapters 2 and 3, a developer could produce performance models and use them to engineer the performance of parallel applications. Indeed, the data presented in these chapters, and the accompanying models of operation performance derived from them, could be used directly to predict the performance of simple scientific applications on the Cray T3D or on networked workstations. Completely symbolic models could be produced by combining the algebraic expressions for execution time of individual component operations. This approach represents an advance over ad hoc manual model construction, which is current practice.

In Chapter 4 we turned our attention to symbolic models of applications' execution times. Developing models by hand, we described the performance of collective communications operations in more detail than in Chapter 3, enabling different regimes of parameter space to be identified. We combined these models to form expressions for the execution time of complete applications. We saw how such models could be used to tune the parameters of a particular implementation, to compare alternative implementations, or to compare alternative platforms. We used a case analysis technique to distinguish different regimes of a model's parameter space and remove min and max operators from expressions. This potentially leads to a large number of regimes which

must be considered independently. More critically, we also encountered difficulties with reasoning informally about complex patterns of interactions between processes, and were forced to resort to loose bounds on execution time.

A major contribution of this thesis is the novel framework for reasoning about concurrent system performance developed in Chapter 5. We define the syntax and semantics of the Eager Timed Calculus of Communicating Systems, and define a notion of execution time for models expressed in this language. Most timed process calculi are designed for reasoning about time-dependent behavioural properties of concurrent systems. Eager Timed CCS can be used to reason about these properties, but it is specifically designed for reasoning about the performance of concurrent systems. The calculus extends previous work in this area by including eagerness and a notion of execution which underlies the derivation of execution times from models. We provide a rigorous methodology for analysing the performance of concurrent systems, allowing the performance of a parallel system to be derived from a formal description of the system's components. Our approach differs from previous work on timed process calculi for performance modelling, which have relied on stochastic distributions to model time, and traditional Markov analysis to evaluate model performance.

The most widely used performance evaluation technique for parallel applications is the measurement of an implementation's performance on the target platform. Alternative designs must be implemented in order to be evaluated. Our approach, like all predictive modelling approaches, allows the performance of alternative designs to be compared without each design being implemented. In contrast with queueing network models, our approach can readily capture condition synchronisation. Recent work [Adve and Vernon, 1993] criticises the widespread use of the exponential distribution in the stochastic analysis of parallel systems and argues that deterministic models are adequate for performance prediction. Unlike stochastic modelling approaches, which model the execution times of component operations as statistical distributions, our approach models basic execution times deterministically.

Our methodology has been automated in an analysis tool, which we used to explore the

performance of models of parallel applications in Chapter 6. The analysis tool allows us to investigate systems of greater complexity than we can viably analyse informally. The tool's method of analysis is similar to a discrete event simulation of the model. Although the analysis tool expands the range of systems which we can analyse, it is limited to small models by its large resource requirements.

Other formalisms have been used to analyse much larger systems than we can currently handle in Eager Timed CCS. Of these, timed Petri net approaches are arguable the most mature, in terms of both efficient analysis techniques and the availability of sophisticated tools. Nevertheless, these tools are not widely used by parallel application developers. We contend that Eager Timed CCS, with its basic concepts of process and communication, is a more natural formalism for developers to describe parallel application. We also believe that it is a natural formalism in which to explore practically important behavioural correctness properties of parallel systems, such as absence of deadlock. An integrated behavioural and performance formalism may be attractive to developers. However, improvements in the underlying analysis theory, its implementation in the analysis tool, and the integration of the analysis tool with common parallel programming tools will all be required before it can be applied to engineering the performance of parallel applications.

Further improvements to the efficiency of analysis could be achieved through exploiting symmetries in the models. Parallel applications typically have very regular structures involving a number of similar processes. Hence, models of parallel applications often exhibit symmetries. Models of parallel applications in Eager Timed CCS often generate state spaces which include pairs of syntactically distinct but behaviourally equivalent states. Since the analysis engine relies on traversing the state space, modifications which would recognise equivalent states could lead to significant performance improvements, enabling the analysis of larger models. A syntactic equality test which recognised permutations of processes in parallel compositions could be effective. A behavioural equality test based on bisimulation could offer larger reductions in the state space representation, but at greater computational cost. Performance improvements are necessary to enable the efficient analysis of larger models.

The analysis tool allows the execution time of a model to be derived at a particular point in parameter space. If we wished to explore a system's performance across parameter space, we would be forced to sample the parameter space by performing a number of "concrete" analyses at different parameter settings. In general, this will involve significant repetition of work. In order to predict performance at other parameter settings, we would have to perform further analyses, or fit a model to the data. During each analysis, we derive much of the information required to characterise different regimes of parameter space. This information could be used to provide detailed symbolic models of the system's execution time, but is not available with the model-fitting approach. To address these weaknesses, we enhanced the semantics of Eager Timed CCS to cope with more abstract, parametric agents in Chapter 7. A new analysis tool, drawing heavily on constraint logic programming techniques, was implemented to automate the analysis of parametric agents. This tool can partition the parameter space of a model into regimes in which execution time is characterised by a single expression, which the tool derives automatically. The tool has been demonstrated on very small models, but the resource requirements of the current implementation preclude the analysis of worthwhile examples.

Many stochastic modelling techniques use numerical methods to solve an underlying stochastic process, or use simulation techniques to produce a numerical measure of performance. Like the "concrete" Eager Timed CCS methodology, these approaches do not readily support the development of a symbolic expression for execution time. They rely on the designer to provide a model of the appropriate form which can then be fitted to data derived from a number of analyses of model instances at different points in parameter space. Our parametric approach exploits information which is implicit in the model to derive a set of symbolic expressions characterising execution time in different regions of parameter space.

The applicability of the parametric analysis tool could be significantly extended by a more efficient implementation. In particular, a fully incremental constraint solver would allow larger models to be more efficiently analysed. However, it is not clear that this will enable the analysis of models which are large enough to describe realistic parallel

applications. As the complexity of models grow, the complexity of the analysis results also increases, and symbolic simplification techniques will be required to present results in a form which is amenable to interpretation by the user.

For performance engineering to become widely adopted in parallel application design, efficient, intuitive and well-integrated tools will be required. The techniques and tools presented in this thesis are small steps in interesting directions towards that goal, but much remains to be done to bridge the gap between performance engineering and practical activity in parallel software development.

Bibliography

[Adve and Vernon, 1993] V. Adve and M. Vernon. The influence of random delays on parallel execution times. *Performance Evaluation Review*, 21(1):61–73, June 1993. Proceedings of ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems.

[Aggrawal *et al.*, 1989] A. Aggrawal, A. K. Chandra, and M. Snir. On communication latency in PRAM computatations. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[Alt *et al.*, 1987] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal of Computing*, 16(5):808–835, 1987.

[Appel *et al.*, 1992] A. W. Appel, J. S. Mattson, and D. R. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, October 1992.

[AT&T Bell Laboratories, 1993] AT&T Bell Laboratories. The standard ml of new jersey library reference manual (release 0.1). Distributed with Standard ML of New Jersey, February 1993.

[Baeten and Bergstra, 1991] J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[Bal *et al.*, 1989] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261, September 1989.

[Balasundaram *et al.*, 1990] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. Technical Report CRPC-TR90093, Center for Research on Parallel Computation, Rice University, Houston, Texas, U.S.A., October 1990.

[Beguelin *et al.*, 1991] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM: Parallel Virtual Machine. Technical Report TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tennessee, U.S.A., 1991.

[Benker *et al.*, 1992] S. Benker, B. M. Chapman, and H. P. Zima. Vienna Fortran 90. Technical Report ACPC/TR 92-15, Department for Statistics and Computer Science, University of Vienna, Austria, September 1992.

[Bergstra and Klop, 1985] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[Berry *et al.*, 1989] M. Berry *et al.* The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[Bomans *et al.*, 1990] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[Boyle *et al.*, 1987] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, 1987.

[Brehm *et al.*, 1995] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy. PerPreT — A Performance Prediction Tool for Massively Parallel Systems. Performance Evaluation Group, Vanderbilt University School of Engineering, Nashville, Tennessee, U.S.A., 1995.

[Brehm, 1995] J. Brehm. The LOOP Approach, a new Method for the Evaluation of Parallel Systems. Performance Evaluation Group, Vanderbilt University School of Engineering, Nashville, Tennessee, U.S.A., 1995.

[Brémond-Grégoire, 1994] P. Brémond-Grégoire. *A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, U.S.A., May 1994.

[Brewer *et al.*, 1992] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. *Performance Evaluation Review*, 20(1):247–248, June 1992. Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[Bruce *et al.*, 1995] R. A. A. Bruce, S. Chapple, N. B. MacDonald, A. S. Trew, and S. Trewin. CHIMP and PUL: Support for portable parallel computing. *Future Generation Computer Systems (FGCS)*, 11(1), January 1995.

[Campbell and Turner, 1994] D. K. G. Campbell and S. J. Turner. CLUMPS: a model of efficient, general purpose parallel computation. Technical report, Department of Computer Science, University of Exeter, Exeter, U.K., 1994.

[Candlin *et al.*, 1992] R. Candlin, P. Fisk, J. Phillips, and N. Skilling. Studying the performance properties of concurrent programs by simulation experiments on synthetic programs. *Performance Evaluation Review*, 20(1):239–240, June 1992. Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[Caselli *et al.*, 1994] S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on SIMD massively parallel GSPN analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools — 7th International Conference*, pages 266–283. Springer-Verlag, 1994. Lecture Notes in Computer Science 794.

[Chen, 1993] L. Chen. *Timed Processes: Models, Axioms and Decidability.* PhD thesis, Department of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K., June 1993.

[Childers *et al.*, 1995] C. A. Childers, A. W. Apon, W. H. Hooper, K. D. Gordon, and L. W. Dowdy. The Multigraph Modeling Tool. In *7th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, U.S.A., October 1995.

[Choi *et al.*, 1994] J.-Y. Choi, I Lee, and I. Kang. Timing analysis of superscalar processor programs using ACSR. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[Clarke *et al.*, 1995] D. Clarke, I. Lee, and H.-L. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer Software Engineering*, 1995. To appear.

[Cleaveland *et al.*, 1989] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems*. Springer-Verlag, 1989. Lecture Notes in Computer Science 407.

[Cleaveland *et al.*, 1993] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Cohen and Weitzman, 1992] J. Cohen and A. Weitzman. Software Tools for Microanalysis of Programs. *Software — Practice and Experience*, 22(9):777–808, September 1992.

[Cohen, 1982] J. Cohen. Computer-Assisted Microanalysis of Programs. *Communications of the ACM*, 25(10):724–733, October 1982.

[Cole, 1989] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, London, UK, 1989. Research Monographs in Parallel and Distributed Computing.

[Culler *et al.*, 1993] D. Culler *et al.* LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[Dantzig *et al.*, 1954] G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality constraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1954.

[Darlington *et al.*, 1993] J. Darlington, M. Ghanen, and H. W. To. Structured Parallel Programming. In *Working Conference on Massively Parallel Programming Models: Suitability, Realization, and Performance*, Berlin, Germany, September 20–23, 1993. IEEE Computer Society Press.

[Davies and Schneider, 1989] J. Davies and S. Schneider. An introduction to Timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, Oxford, U.K., 1989.

[de Ronde *et al.*, 1994] J. F. de Ronde, B. van Halderen, A. de Mes, M. Beemster, and P. M. A. Sloot. Automatic Performance Estimations Of SPMD Programs On MPP. In L. Dekker, W. Smit, and J. C. Zuidervaart, editors, *Massively Parallel Processing Applications and Development*, pages 381–388. Elsevier Science B. V., 1994.

[Dixit, 1991] K. M. Dixit. The SPEC benchmarks. *Parallel Computing*, 17:1195–1210, 1991.

[Dongarra *et al.*, 1993] J. Dongarra, A. Hofer, and E. Strohmaier. Top500 supercomputers, July 1993. `http://parallel.rz.uni-mannheim.de/top500/top500.html`.

[Dongarra *et al.*, 1995] J. Dongarra, A. Hofer, and E. Strohmaier. Top500 supercomputers, July 1995. `http://parallel.rz.uni-mannheim.de/top500/top500.html`.

[Dongarra, 1990] J. J. Dongarra. The LINPACK benchmark: An explanation. In A. J. Van der Steen, editor, *Evaluation of Supercomputers*, pages 1–21. Chapman and Hall, London, 1990.

[Driscoll and Daasch, 1995] M. A. Driscoll and W. R. Daasch. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25:16–30, 1995.

[Dunlop and Hey, 1993] A. N. Dunlop and A. J. G. Hey. Results of applying statement benchmarks for estimating the execution time of parallel programs. Department of Electronics and Computer Science, University of Southampton, Southampton, U.K., December 1993.

[Eager *et al.*, 1989] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computer Systems*, 38(3):408–423, March 1989.

[Fahringer, 1993] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Department of Software Technology and Parallel Systems, University of Vienna, Austria, 1993.

[Fahringer, 1994] T. Fahringer. The Weight Finder: an advanced profiler for Fortran programs. In C. W. Keßler, editor, *Automatic Parallelization, New Approaches to Code Generation, Data Distribution and Performance Prediction.* Vieweg Verlag, Wiesbaden, Germany, 1994.

[Feo, 1988] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing*, 7:163–185, 1988.

[Ferscha and Chiola, 1994] A. Ferscha and G. Chiola. Accelerating the evaluation of parallel performance models using distributed simulation. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools — 7th International Conference*, pages 231–252. Springer-Verlag, 1994. Lecture Notes in Computer Science 794.

[Ferscha, 1992] A. Ferscha. A Petri Net Approach for Performance Oriented Parallel Program Design. *Journal of Parallel and Distributed Computing*, 15(3):188–206, July 1992.

[Flynn, 1972] M. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[Fortune and Wyllie, 1978] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[Geist *et al.*, 1990] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A Portable Instrumented Communication Library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, Tennessee, U.S.A., July 1990.

[Geist *et al.*, 1994] A. Geist, A. Beguelin, J. Dongarra, W. Hiang, R. Manchek, and V. Sunderam. *PVM:Parallel Virtual Machine — A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[Getov and Hockney, 1993] V. S. Getov and R. W. Hockney. Comparative performance analysis of uniformly distributed applications. In *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*, 1993.

[Getov *et al.*, 1993] V. S. Getov, R. W. Hockney, and A. J. G. Hey. Performance analysis of distributed applications by suitability functions. In *Working Conference on Massively Parallel Programming Models: Suitability, Realization, and Performance*, Berlin, Germany, September 20–23, 1993. IEEE Computer Society Press.

[Gilmore and Hillston, 1994] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In G. Haring, editor, *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 1994.

[Gorrieri and Rocetti, 1993] R. Gorrieri and M. Rocetti. Towards performance evaluation in process algebras. In *Proceedings of 3rd International Conference on Algebraic Methodology and Software Technology*, 1993.

[Gustafson, 1988] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Harrison, 1991] R. J. Harrison. Portable tools and applications for parallel computers. *Internation Journal of Quantum Chemistry*, 40(847), 1991.

[Hatcher and Quinn, 1991] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[Hempel *et al.*, 1992] R. Hempel, H.-C. Hoppe, and A. Supalov. PARMACS 6.0 Library Interface Specification. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, Germany, December 1992.

[Hermanns *et al.*, 1994] H. Hermanns, V. Mertsiotakis, and M. Rettelbach. Performance analysis of distributed systems using TIPP — a case study. In J. Hillston and

R. Pooley, editors, *Proceedings of the 10th UK Performance Engineering Workshop for Computer and Telecommunications Systems*, 1994.

[Hey, 1991] A. J. G. Hey. The Genesis distributed-memory benchmarks. *Parallel Computing*, 17:1275–1283, 1991.

[Heywood and Ranka, 1992] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation — I. The Model. *Journal of Parallel and Distributed Computing*, 16(3):212–232, November 1992.

[Hickey *et al.*, 1992] T. J. Hickey, J. Cohen, H. Hirofumi, and T. Petitjean. Computer-Assisted Microanalysis of Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 14(1):54–106, January 1992.

[Hillston, 1994] J. E. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Department of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom, 1994.

[Hiranandani *et al.*, 1991] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hockney and Berry, 1994] R. W. Hockney and M. Berry. Public international benchmarks for parallel computers. *Scientific Programming*, 3:101–146, 1994.

[Hockney and Curington, 1989] R. W. Hockney and I. J. Curington. $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10:277–286, 1989.

[Hockney, 1987] R. W. Hockney. Parametrization of computer performance. *Parallel Computing*, 5:97–103, 1987.

[Hockney, 1991] R. W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17:1111–1130, 1991.

[Jonkers, 1994a] H. Jonkers. Probabilistic performance modelling of parallel numerical applications: a case study. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *Parallel Computing: Trends and Applications*, pages 609–612. North Holland, 1994. Proceedings of the International Conference ParCo93.

[Jonkers, 1994b] H. Jonkers. Queueing Models of Parallel Applications: The Glamis Methodology. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools — 7th International Conference*, pages 123–138. Springer-Verlag, 1994. Lecture Notes in Computer Science 794.

[Kapelnikov *et al.*, 1989] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. Modeling methodology for the analysis of concurrent systems and computations. *Journal of Parallel and Distributed Computing*, 6:568–597, 1989.

[Kapelnikov *et al.*, 1992] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A methodology for performance analysis of parallel computations with looping constructs. *Journal of Parallel and Distributed Computing*, 14:105–120, 1992.

[Kennedy *et al.*, 1991] K. Kennedy, N. McIntosh, and K. S. McKinley. Static Performance Estimation in a Parallelizing Compiler. Technical Report TR91-174, Rice University, Department of Computer Science, P.O. Box 1892, Houston, Texas 77251, U.S.A., December 1991.

[Kitajima *et al.*, 1993] J. P. Kitajima, C. Tron, and B. Plateau. ALPES: a tool for the performance evaluation of parallel programs. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 213–228. North-Holland, Amsterdam, 1993.

[Kumar and Rao, 1987] V. Kumar and V. N. Rao. Parallel depth-first search. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[Lassez and McAloon, 1988] J. L. Lassez and K. McAloon. Applications of a canonical form for generating linear constraints. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, December 1988.

[Lee *et al.*, 1994] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-based real-time systems. *Proceedings of the IEEE*, 82(1):158–171, January 1994.

[Lin and Snyder, 1992] C. Lin and L. Snyder. The Kheystone Benchmark for Parallel Performance Prediction. In E. D. Brooks, B. J. Heston, K. H. Warren, and L. J. Woods, editors, *The 1992 Massively Parallel Computing Initiative Yearly Report: Harnessing the Killer Micros*. Lawrence Livermore National Laboratory, California 94550, U.S.A., August 1992.

[MacDonald and Fletcher, 1994] N. B. MacDonald and R. A. Fletcher. Portability and Code Reuse in Parallel Applications. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *Parallel Computing: Trends and Applications*, pages 609–612. North Holland, 1994. Proceedings of the International Conference ParCo93.

[MacDonald and Trew, 1994] N. B. MacDonald and A. S. Trew, editors. *The State of the Art in Cluster Computing — Proceedings of the JISC/NTSC Workshop November 1993*, The University of Edinburgh, Edinburgh, U. K., February 1994. Edinburgh Parallel Computing Centre Technical Report EPCC-TR94-07.

[MacDonald, 1994] N. B. MacDonald. Predicting the Performance of Sequential Scientific Codes. In C. W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*. Vieweg Verlag, 1994.

[MacDonald, 1995] N. B. MacDonald. A framework for portable parallel applications. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, chapter 13, pages 225–242. Oxford University Press, 1995.

[Mak and Lundstrom, 1990] V. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *Transactions on Parallel and Distributed Systems*, 1(3):257–270, 1990.

[Malony *et al.*, 1994] A. D. Malony, V. Mertsiotakis, and A. Quick. Automatic scalability analysis of parallel programs based on modelling techniques. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools — 7th International Conference*, pages 123–138. Springer-Verlag, 1994. Lecture Notes in Computer Science 794.

[McMahon, 1986] F. H. McMahon. L.L.N.L FORTRAN kernels: MFLOPS. Lawrence Livermore National Laboratory, California, U.S.A., 1986.

[Mendes, 1993] C. L. Mendes. Performance Prediction by Trace Transformation. In *Fifth Brazilian Symposium on Computer Architecture*, Florianópolis, September 1993. Department of Computer Science, University of Illinois, Urbana, Illinois, U.S.A.

[Milner, 1980] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Milner, 1983] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[Moller and Tofts, 1990] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of CONCUR 90*, pages 401–415. Springer Verlag, 1990. Lecture Notes in Computer Science 458.

[Moncrieff *et al.*, 1995] D. Moncrieff, R. E. Overill, and S. Wilson. α_{critical} for parallel processors. *Parallel Computing*, 21:467–471, 1995.

[Norman and Thanisch, 1991] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. Technical Report EPCC-TR91-14, Edinburgh Parallel Computing Centre, The University of Edinburgh, Edinburgh, U.K., 1991.

[Pooley, 1995a] R. Pooley. Integrating behavioural and simulation modelling. CSG Report ECS-CSG-8-95, Computer Systems Group, Department of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, U.K., March 1995.

[Pooley, 1995b] R. J. Pooley. *Formalising the Description of Process Based Simulation Models*. PhD thesis, Department of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, U.K., 1995.

[Qin *et al.*, 1991] B. Qin, H. A. Sholl, and R. A. Ammar. Micro time cost analysis of parallel computations. *IEEE Transactions on Computers*, 40(5):613–628, May 1991.

[Reed and Roscoe, 1986] G. M. Reed and W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of the International Conference on Automata, Languages and Programming*. Springer-Verlag, 1986. Lecture Notes in Computer Science 226.

[Reed *et al.*, 1987] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems. *IEEE Transactions on Computers*, C-38(7):845–858, July 1987.

[Reinhardt *et al.*, 1993] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. *Performance Evaluation Review*, 21(1):48–60, June 1993. Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[Sinclair and Dawkins, 1994] J. B. Sinclair and W. P. Dawkins. ES: A tool for predicting the performance of parallel systems. In *Proceedings of MASCOTS 94*, pages 164–168. IEEE Computer Society Press, 1994.

[Singh *et al.*, 1991] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Technical Report CSL-TR-91-469, Stanford University Computer Systems Laboratory, California, U.S.A., 1991.

[Smirni and Rosti, 1994] E. Smirni and E. Rosti. Modeling Speedup of SPMD Applications on the Intel Paragon: A Case Study. In *Proceedings of High Performance Computing and Networking*. Springer-Verlag, 1994.

[Sötz, 1990] F. Sötz. A method for performance prediction of parallel programs. In H. Burkhart, editor, *CONPAR 90—VAPP IV, Joint International Conference on Vector and Parallel Processing*, pages 98–107. Springer-Verlag, 1990.

[Strulo, 1993] B. Strulo. *Process Algebra for Discrete Event Simulation.* PhD thesis, Imperial College, London, U.K., 1993.

[Stuckey, 1990] P. J. Stuckey. A simplex-based algorithm for incremental linear arithmetic constraint solving and detection of implicit equalities. Department of Computer Science, University of Melbourne, Parkville 3052, Australia, 1990.

[Sussman, 1991] A. Sussman. *Model-Driven Mapping of Computation onto Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890, U.S.A., September 1991. Technical Report CMU-CS-91-187.

[Tarditi and Appel, 1991] D. R. Tarditi and A. W. Appel. Ml-yacc user's manual. Distributed with Standard ML of New Jersey, March 1991.

[Transaction Processing Performance Council, 1994] Transaction Processing Performance Council. Summary of TPC Results (As of March 15, 1994). *Performance Evaluation Review*, 21(3&4):7–21, April 1994.

[Tron *et al.*, 1994] C. Tron, Y. Arrouye, J. C. de Kergommeaux, J. P. Kitajima, E. Maillet, B. Plateau, and J.-M. Vincent. Performance evaluation of parallel systems: ALPES environment. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *Parallel Computing: Trends and Applications*, pages 609–612. North Holland, 1994. Proceedings of the International Conference ParCo93.

[Valiant, 1990] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[van Gemund, 1993a] A. J. C. van Gemund. On the analysis of PAMELA models. Technical Report 1-68340-44(1993)05, Laboratory of Computer Architecture and Digital Techniques (CARDIT), Faculty of Electrical Engineering, Delft University of Technology, P.O. Box 5031, NL-2600 GA Delft, The Netherlands, December 1993.

[van Gemund, 1993b] A. J. C. van Gemund. Performance prediction of parallel processing systems: the PAMELA methodology. In *Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, Japan*, July 1993.

[van Gemund, 1993c] A. J. C. van Gemund. The PAMELA Approach to Performance Modeling of Parallel Systems. In *Proceedings of Parallel Computing 93*, 1993.

[Van Hentenryck and Graf, 1992] P. Van Hentenryck and T. Graf. Standard forms for rational linear arithmetic in constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 5:303–320, 1992.

[Van Hentenryck and Imbert, 1993] P. Van Hentenryck and J.-L. Imbert. On the handling of disequations in CLP over linear rational arithmetic. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming : Selected Research*, chapter 4, pages 49–72. The MIT Press, 1993.

[Vrsalovic *et al.*, 1988] D. F. Vrsalovic, D. P. Siewiorek, Z. Z. Segall, and E. F. Gehringer. Performance Prediction and Calibration for a Class of Multiprocessors. *IEEE Transactions on Computers*, 37(11):1353–1365, November 1988.

[Wabnig and Haring, 1994a] H. Wabnig and G. Haring. PAPS — The parallel program performance prediction toolset. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools — 7th International Conference*, pages 284–304. Springer-Verlag, 1994. Lecture Notes in Computer Science 794.

[Wabnig and Haring, 1994b] H. Wabnig and G. Haring. Petri net performance models of parallel systems — methodology and case study. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 301–312. Springer Verlag, 1994. Lecture Notes in Computer Science 817.

[Wagner *et al.*, 1994] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, 1994. To appear.

[Wall, 1990] D. W. Wall. Predicting Program Behaviour Using Real or Estimated Profiles. WRL Technical Note TN-18, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301, U.S.A., December 1990.

[Wang, 1990] K.-Y. Wang. A performance prediction model for parallel compilers. Technical Report CSD-TR-1041, Purdue University, Department of Computer Sciences, West Lafayette, Indiana 47907, U.S.A., November 1990.

[Wang, 1994] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. In *Proceedings of ACM SIGPLAN 94*, pages 73–84, 1994.

[Weiser, 1984] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE 10(4), July 1984.

[Yi *et al.*, 1994] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint solving. In *Proceedings of the 7th International Conference on Formal Description Techniques*, 1994.

[Zemerly *et al.*, 1995] M.J. Zemerly, J. Papay, and Graham R. Nudd. Characterisation based bottleneck analysis of parallel systems. Research Report CS-RR-281, Department of Computer Science, University of Warwick, Coventry, U.K., January 1995.

[Zhang and Xu, 1995] X. Zhang and Z. Xu. Multiprocessor Scalability Predictions Through Detailed Program Execution Analysis. In *9th ACM International Conference on Supercomputing*, July 1995.

[Zima and Chapman, 1990] H. P. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, New York, 1990.